



Implicit Function Ray Tracing

Chris Birke
Secant Astronomy

This talk is about rendering implicit functions, but it's still totally cool if you don't know math or code.

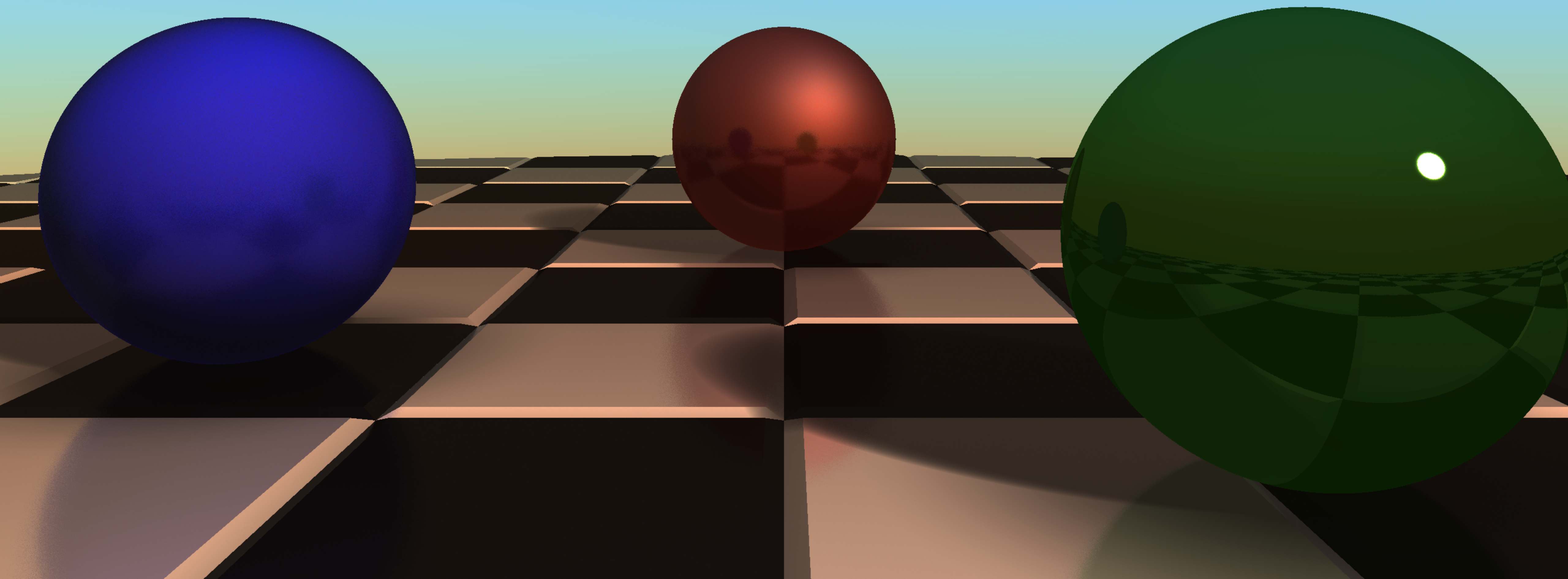
This talk is about art, too.



Live demos to play with.



Ooo Shiny!



A few people to thank.



Iñigo Quílez



Iñigo Quílez

aka iq


$$v(t) = a + bt + ct^2$$

$$a = p = surf(a)$$

Pablo Roman Andrioli



Pablo Roman Andrioli

aka kali

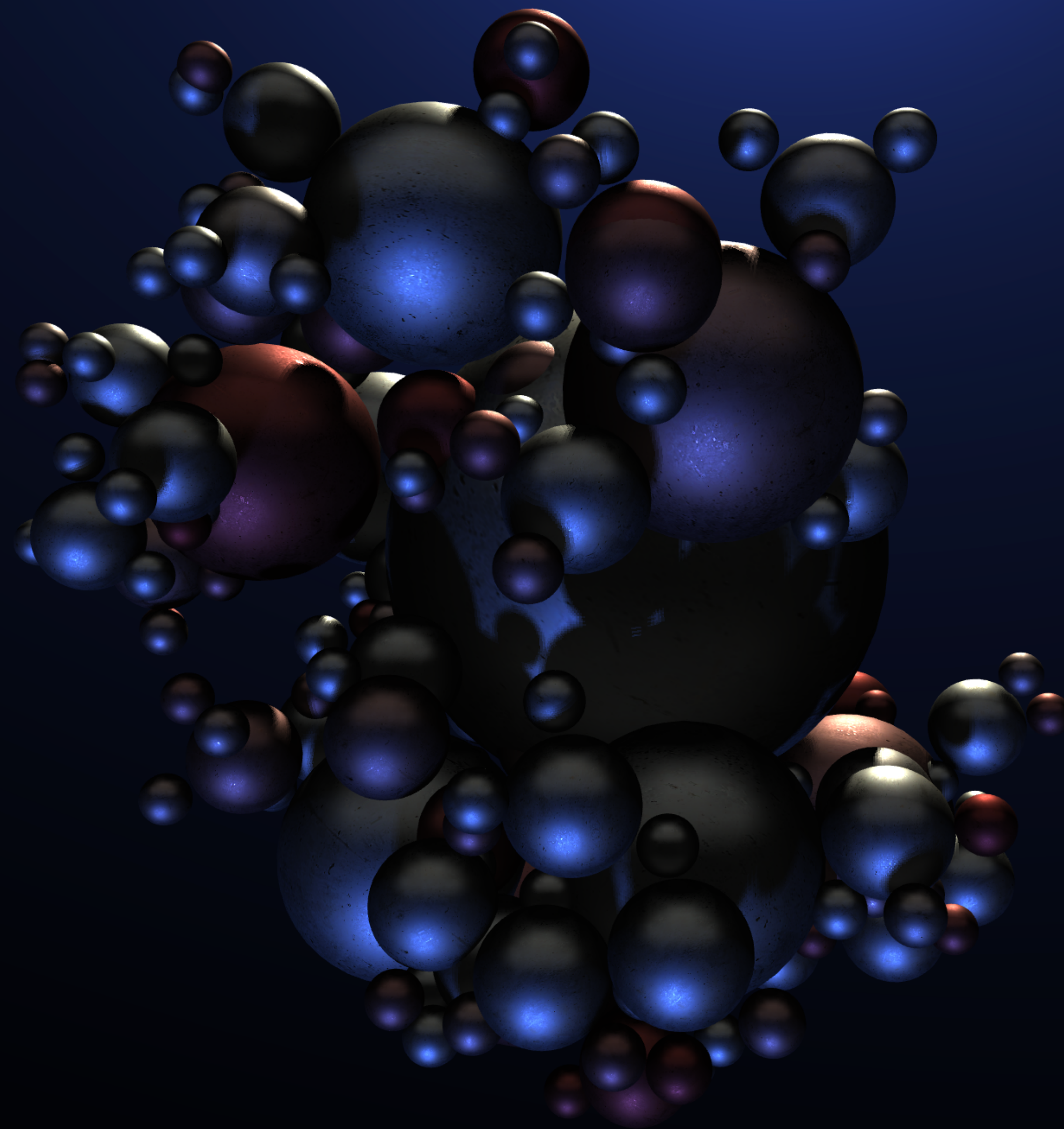


Ricardo Cabello



Ricardo Cabello

aka Mr.doob

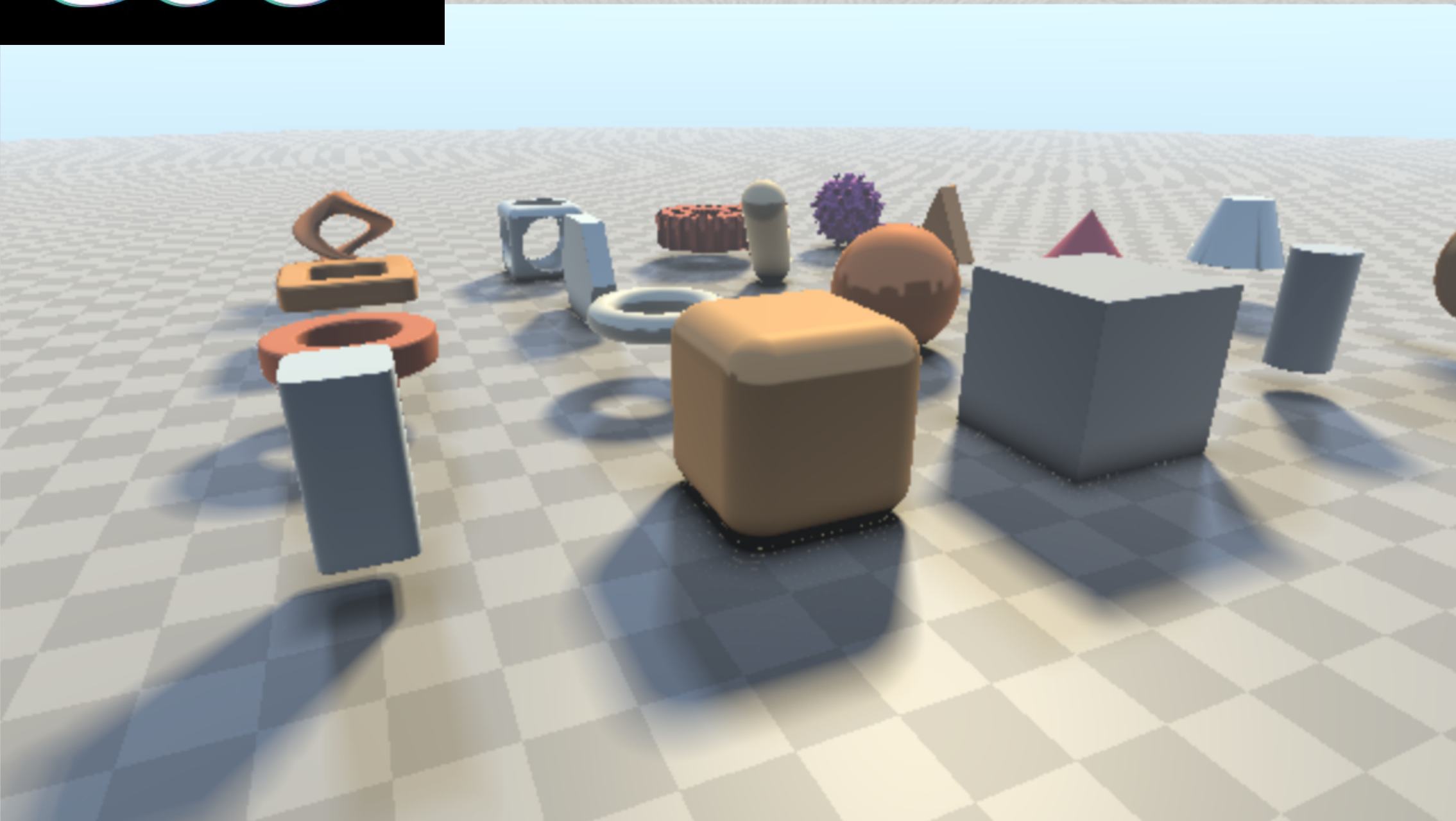


A few places to thank.




```
1 #ifdef GL_ES
2 precision mediump float;
3 #endif
4
5 #extension GL_OES_standard_derivatives : enable
6
7 uniform float time;
8 uniform vec2 mouse;
9 uniform vec2 resolution;
10
11 void main( void ) {
12
13     vec2 position = ( gl_FragCoord.xy / resolution.xy ) + mouse / 4.0;
14
15     float color = 0.0;
16     color += sin( position.x * cos( time / 15.0 ) * 80.0 ) + cos( position.y * cos( time / 15.0 ) * 10.0 );
17     color += sin( position.y * sin( time / 10.0 ) * 40.0 ) + cos( position.x * sin( time / 25.0 ) * 40.0 );
18     color += sin( position.x * sin( time / 5.0 ) * 10.0 ) + sin( position.y * sin( time / 35.0 ) * 80.0 );
19     color *= sin( time / 10.0 ) * 0.5;
20
21     gl_FragColor = vec4( vec3( color, color * 0.5, sin( color + time / 3.0 ) * 0.75 ), 1.0 );
22
23 }
```

GLSL Sandbox



51.99 44.6 fps



Raymarching - Primitives

58441 319 40

Uploaded by iq in 2013-Mar-25

Tags: procedural, 3d, raymarching, distancefields, primitives

A set of primitives and combination functions, for reference. More info here:
<http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>

Comments

Sign in to post a comment.

**hackthechad**, 2016-Feb-21

Awesome reference IQ--learning a lot! Quick question though. What are the bac and dom variables in your lighting calculations?

```
float bac = clamp( dot( nor, normalize(vec3(-lig.x,0.0,-lig.z))), 0.0, 1.0 ) * clamp( 1.0-pos.y,0.0,1.0 );
float dom = smoothstep( -0.1, 0.1, ref.y );
```

**Doublefresh**, 2016-Feb-13

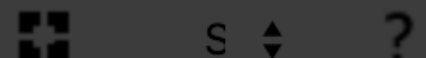
What does the 'c' parameter in sdCone do?

+ Image

Shader Inputs

```
1 // Created by inigo quilez - iq/2013
2 // License Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.
3
4 // A list of usefull distance function to simple primitives, and an example on how to
5 // do some interesting boolean operations, repetition and displacement.
6 //
7 // More info here: http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm
8
9 float sdPlane( vec3 p )
10 {
11     return p.y;
12 }
13
14 float sdSphere( vec3 p, float s )
15 {
16     return length(p)-s;
17 }
18
19 float sdBox( vec3 p, vec3 b )
20 {
21     vec3 d = abs(p) - b;
22     return min(max(d.x,max(d.y,d.z)),0.0) + length(max(d,0.0));
23 }
24
25 float sdEllipsoid( in vec3 p, in vec3 r )
26 {
27     return (length( p/r ) - 1.0) * min(min(r.x,r.y),r.z);
28 }
29
30 float udRoundBox( vec3 p, vec3 b, float r )
31 {
32     return length(max(abs(p)-b,0.0))-r;
33 }
34
35 float sdTorus( vec3 p, vec2 t )
36 {
37     return length( vec2(length(p.xz)-t.x,p.y) )-t.y;
38 }
39
```

6757 chars



Shadertoy

iChannel0

iChannel1

iChannel2

iChannel3

OPENINGS

Techno / Chiptunes

Conspiracy - Binary Flow

Many Cubes

Procedural Graphics

The butterfly effect - Andromeda Software Development

DEALER QUALITY SOFTWARE

THE FLOW

WYNN, WYNN, WYNN, WYNN WYNN



FAIRLIGHT PRESENTS
FULL CONTACT FROM TEAM 17

CRACKED BY GASTON†
SUPPLIED BY DRONE NO. 3

WRITE US NOW :

PO BOX 6, 236 00 HOLLVIKEN, SWEDEN

AAZOO 19XX



REZ

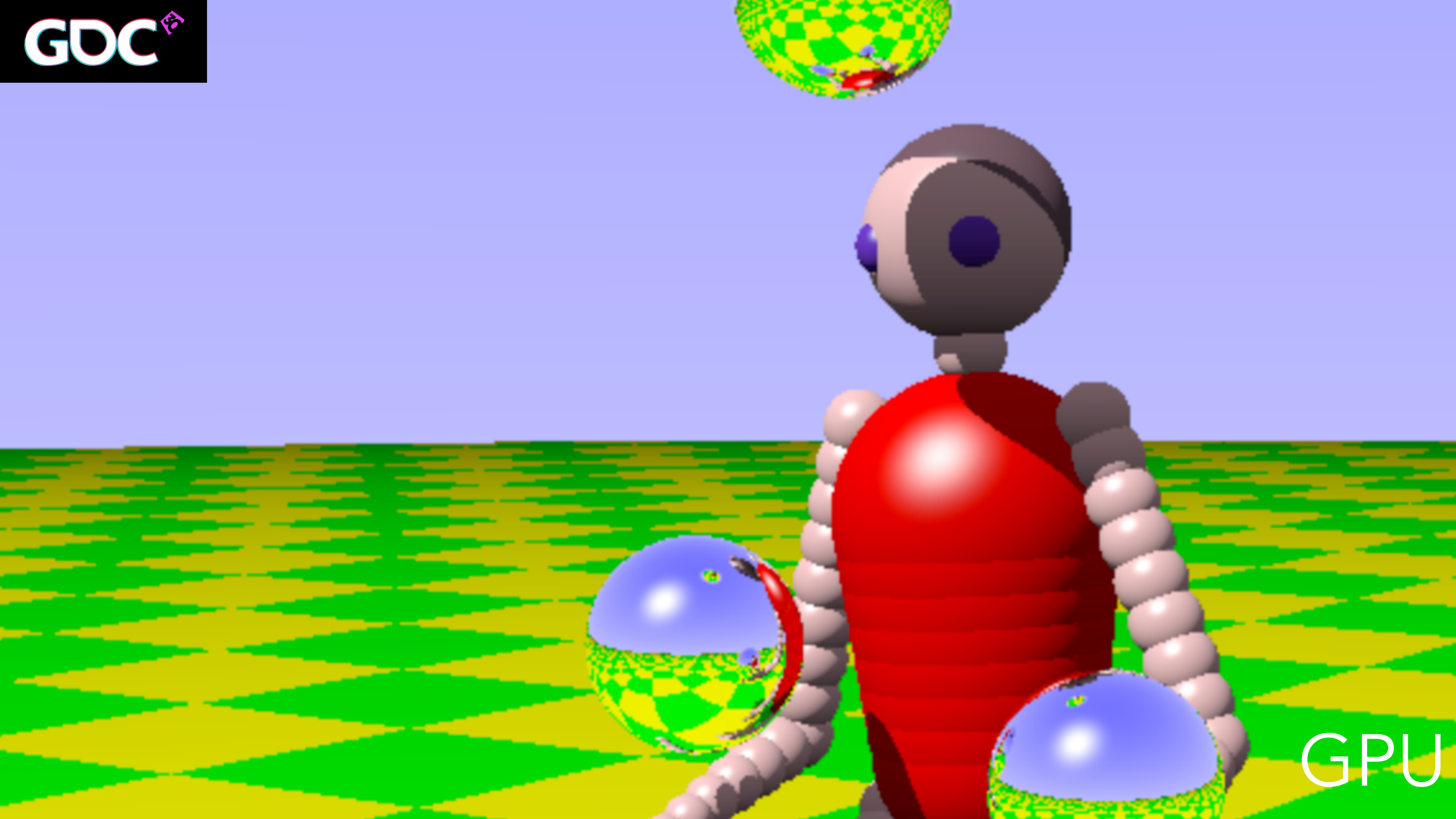
Greetings! Ubisoft • Nice one, bring 'em on!

WARNING! No permanent internet connection needed

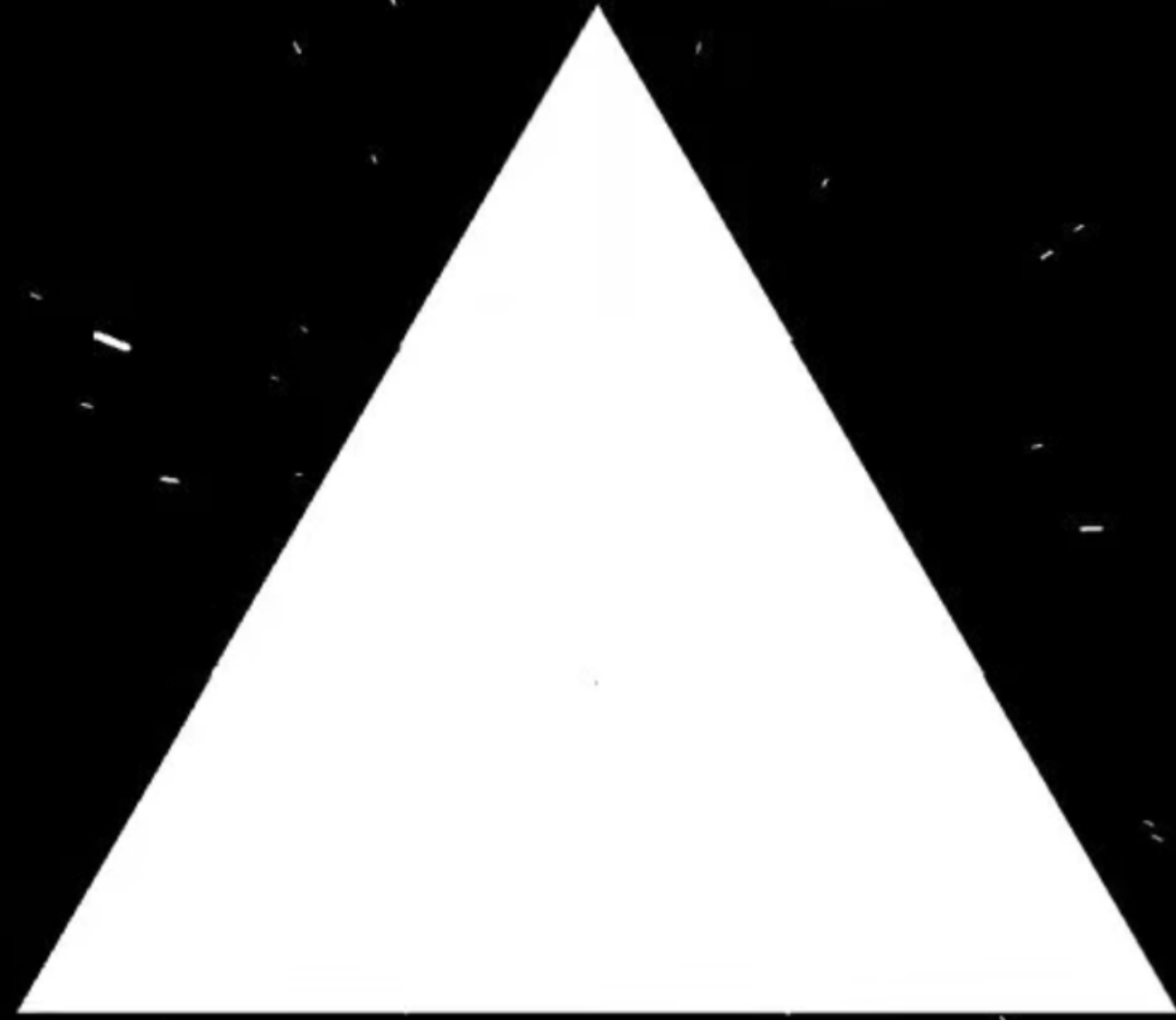




Amiga









Demoscene

Many Cubes

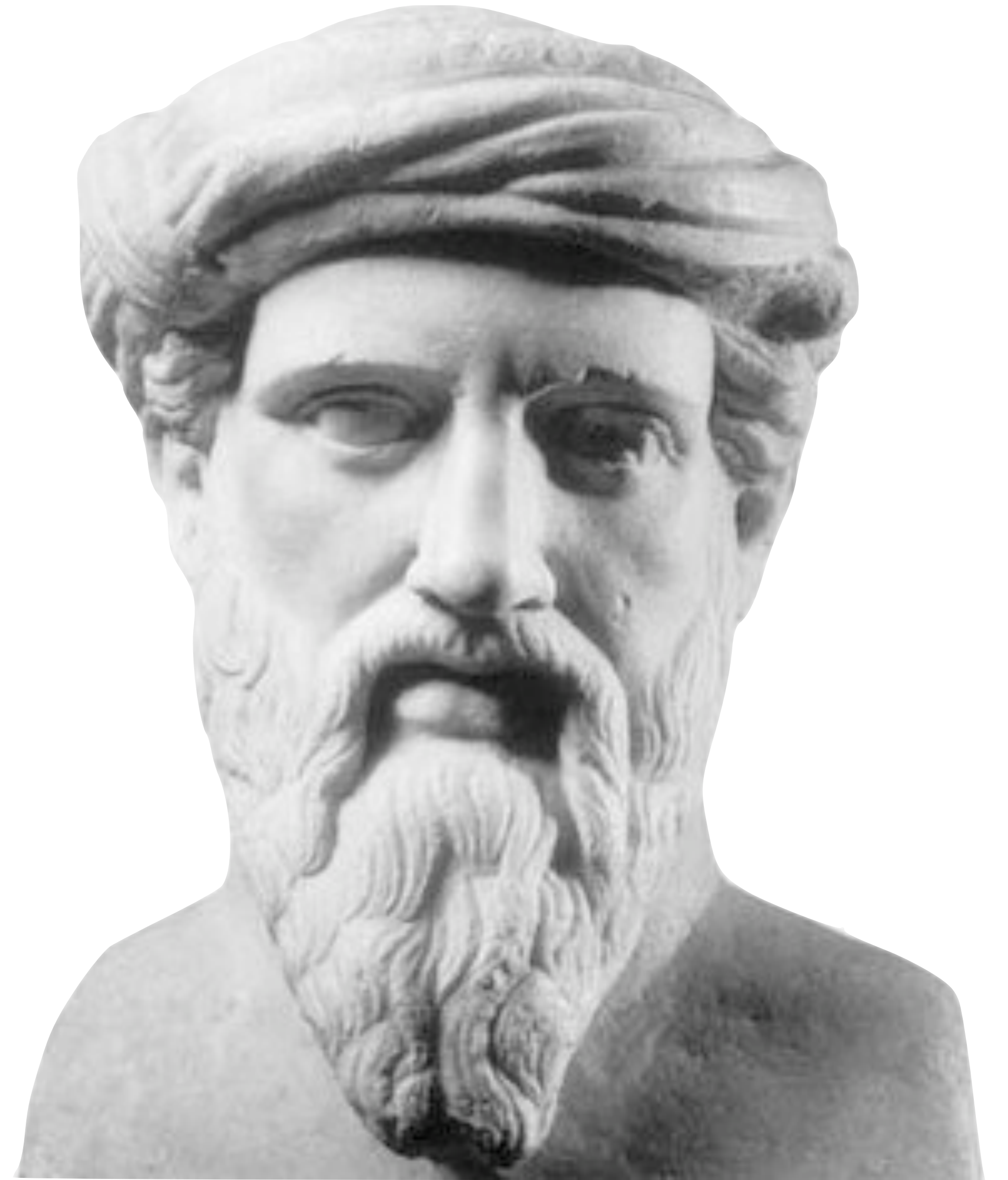
Much Techno

C64/Amiga/GPU

Procedural

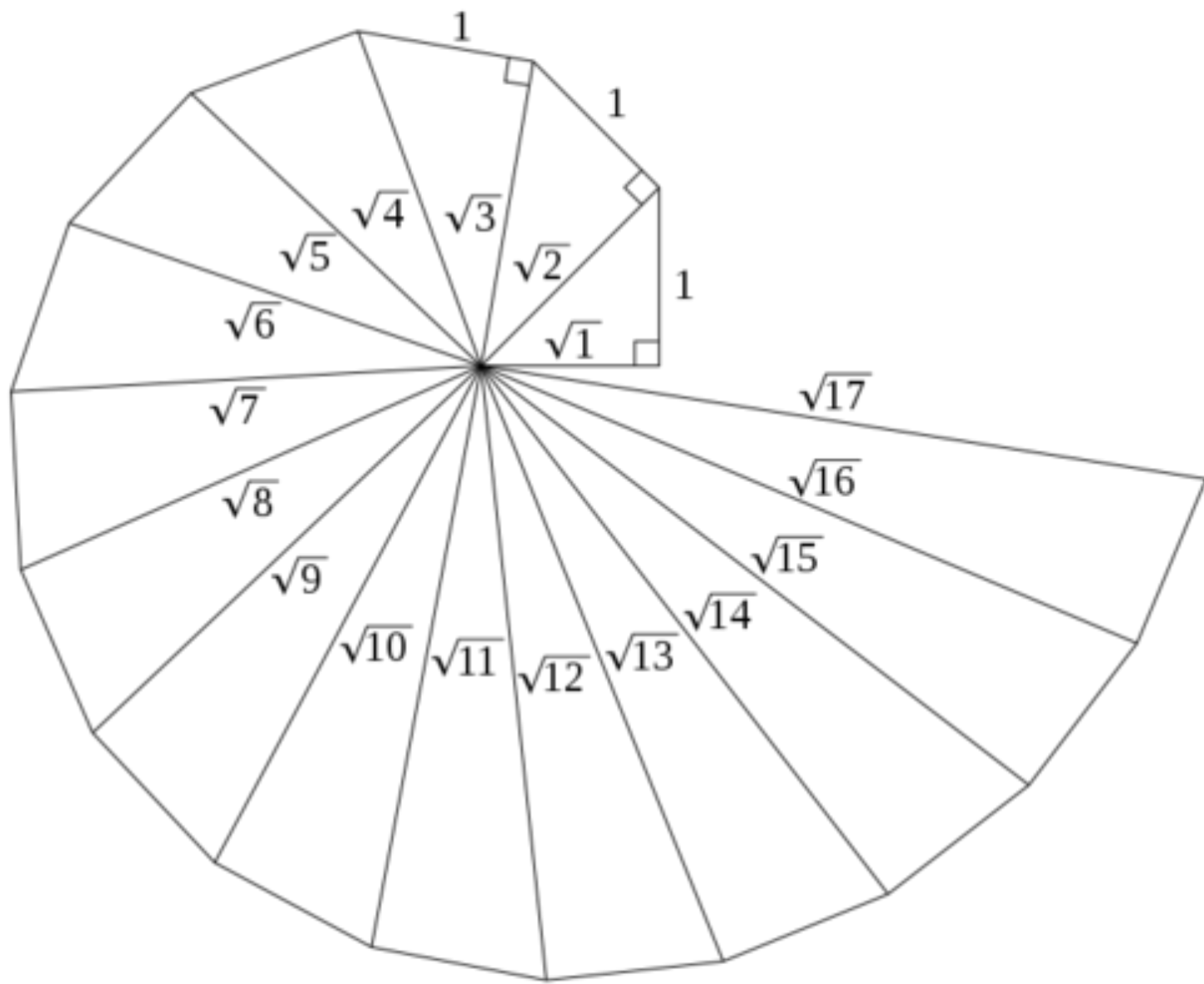
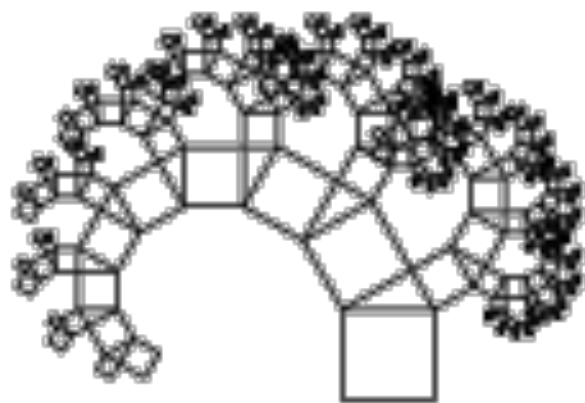
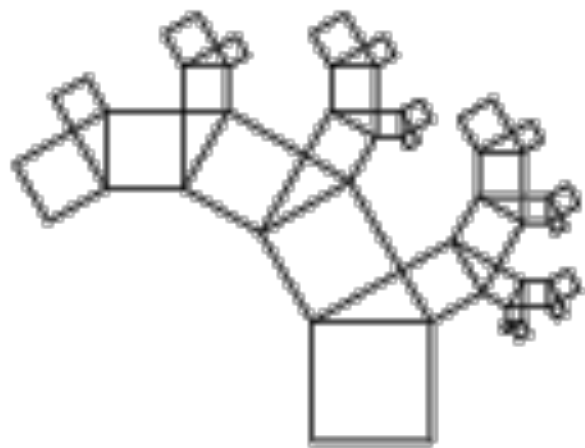
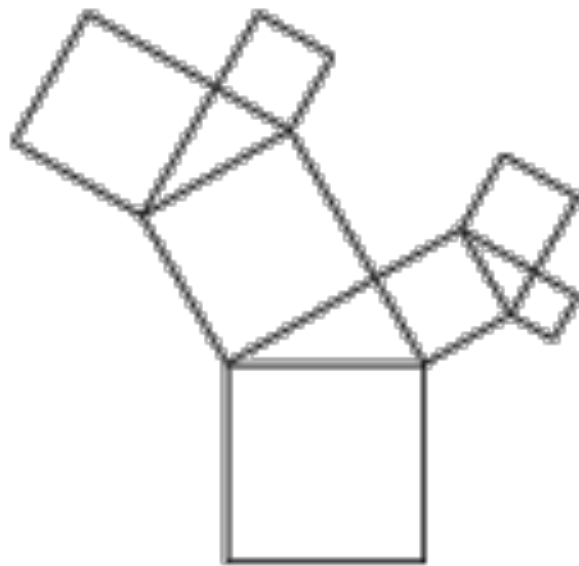
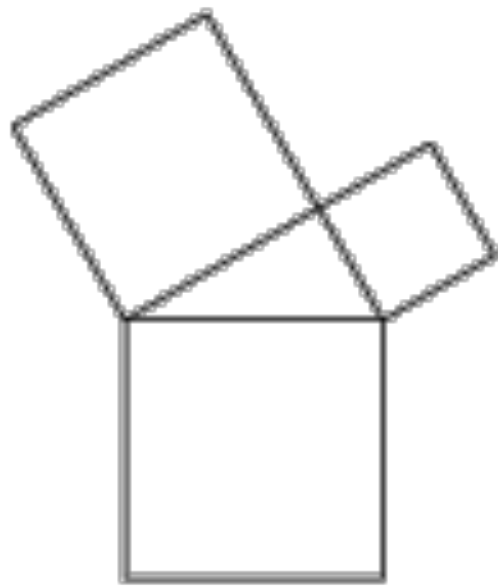
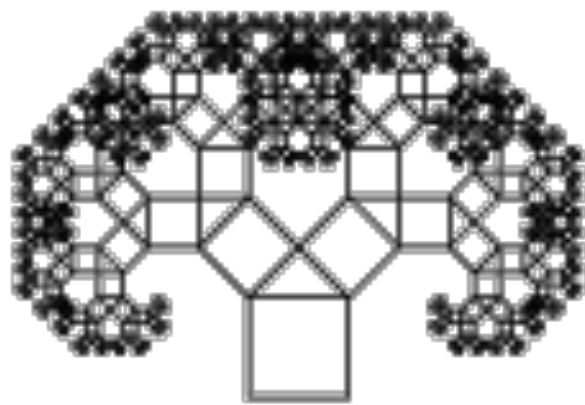
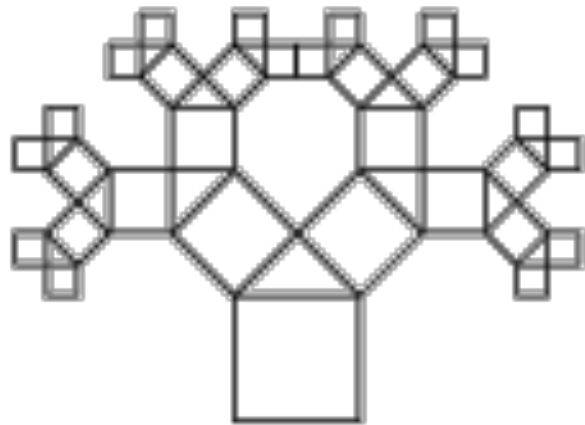
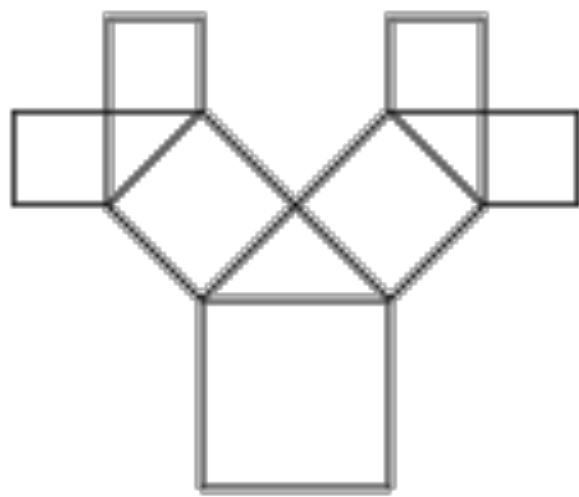
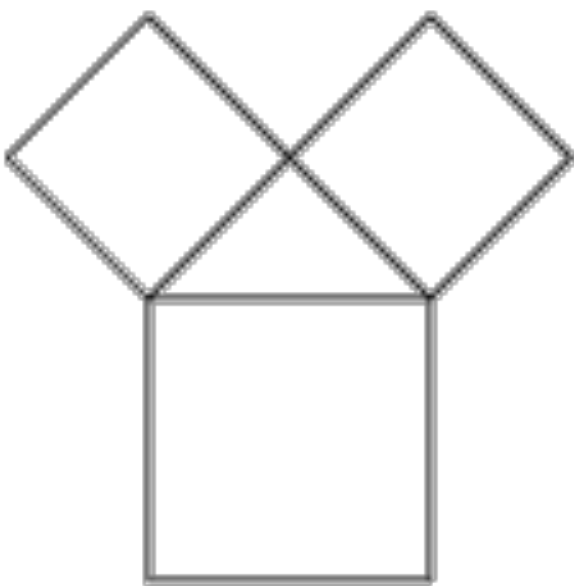
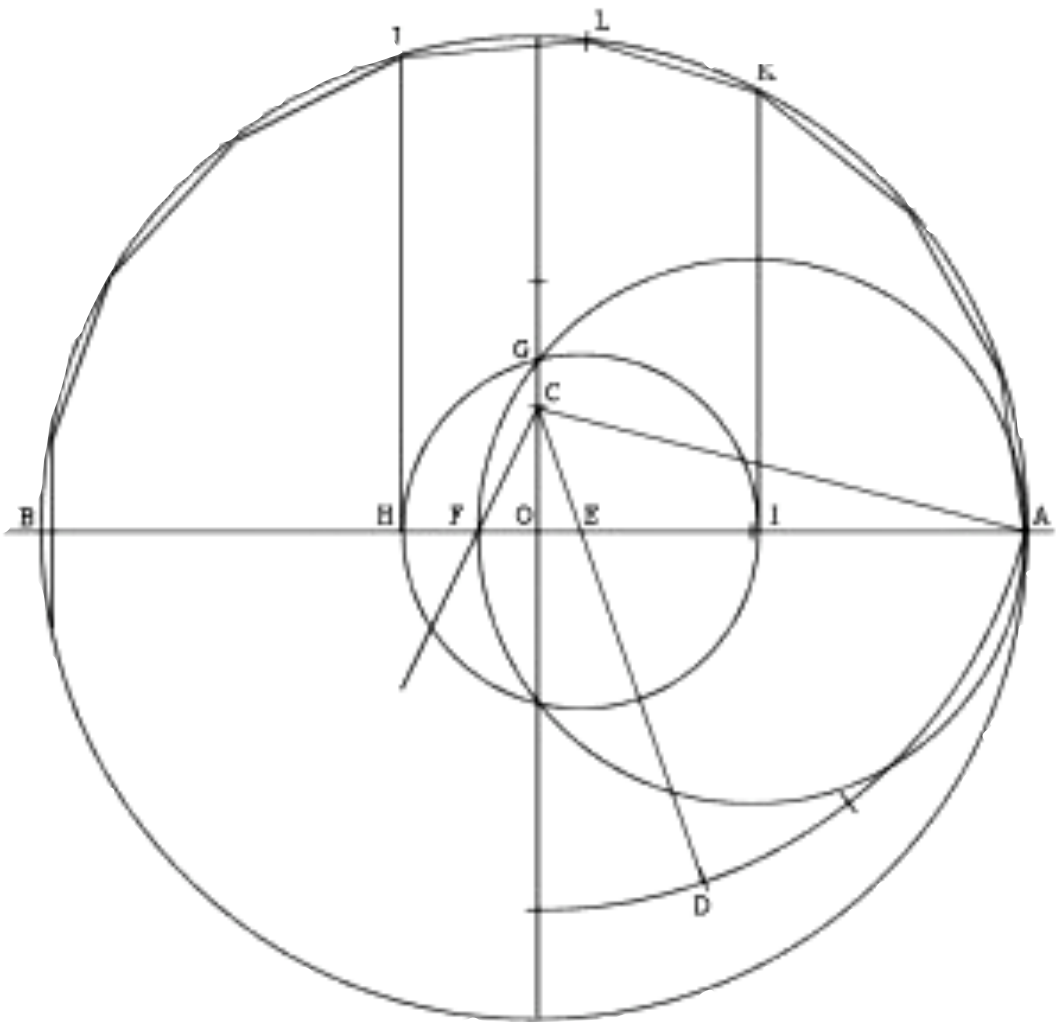
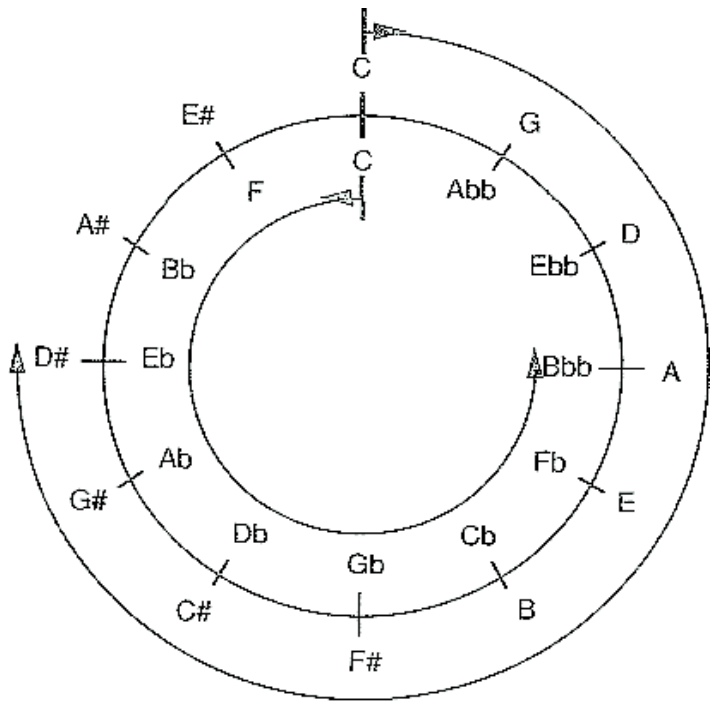
Still Relevant

Πυθαγόρας



Pythagoras

Discovered tons of crazy stuff.





2500 years ago.

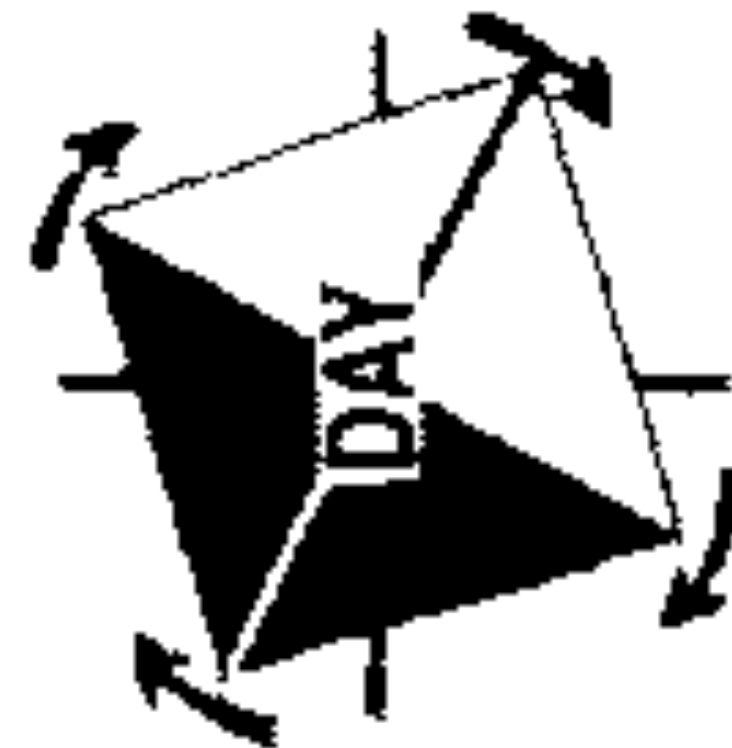
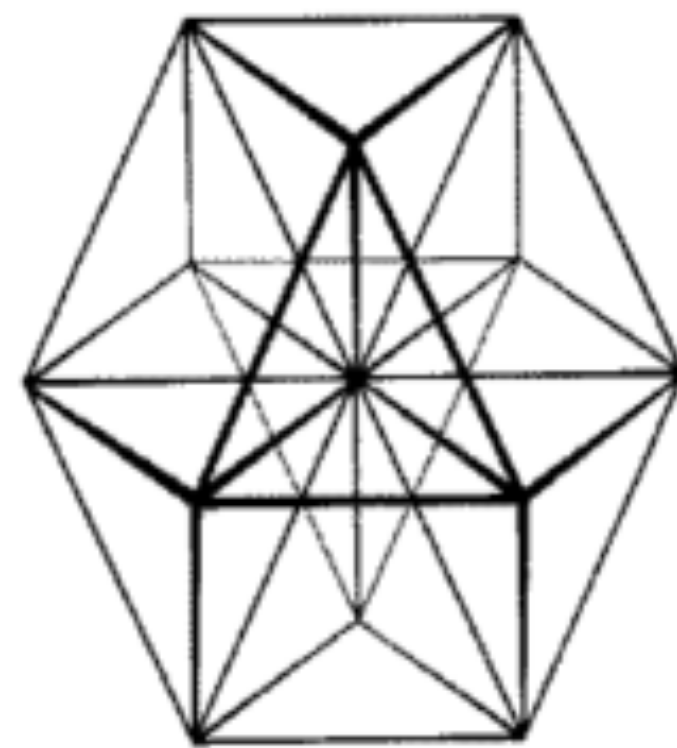
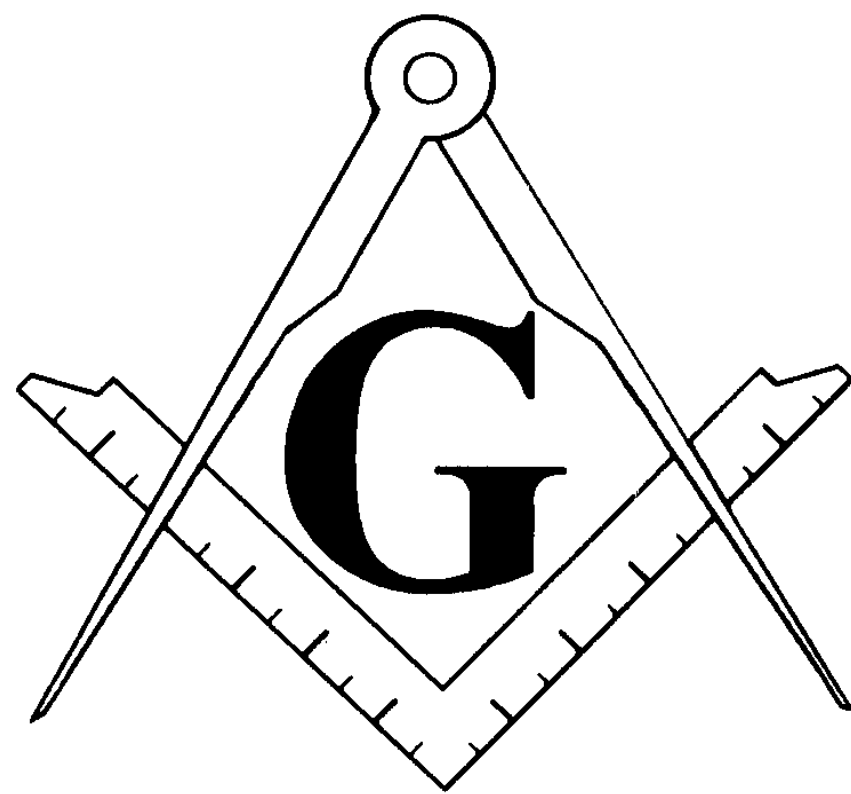
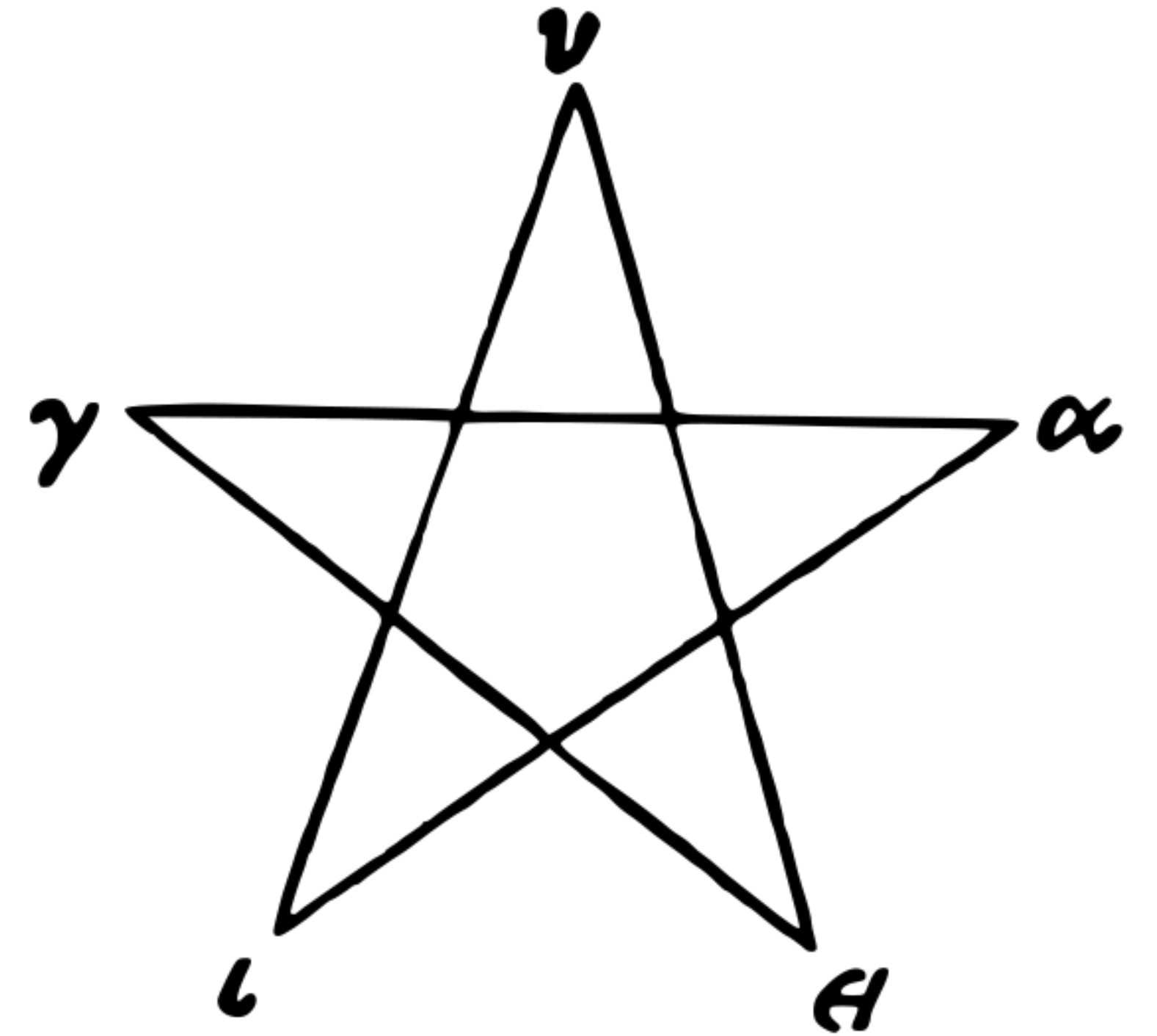
Pythagoras

Straight edge (and compass.)



Pythagoras

Spawned numerous math cults.







.



.

This is the point of the talk.



.

.

Actually, there are two points.

.

.

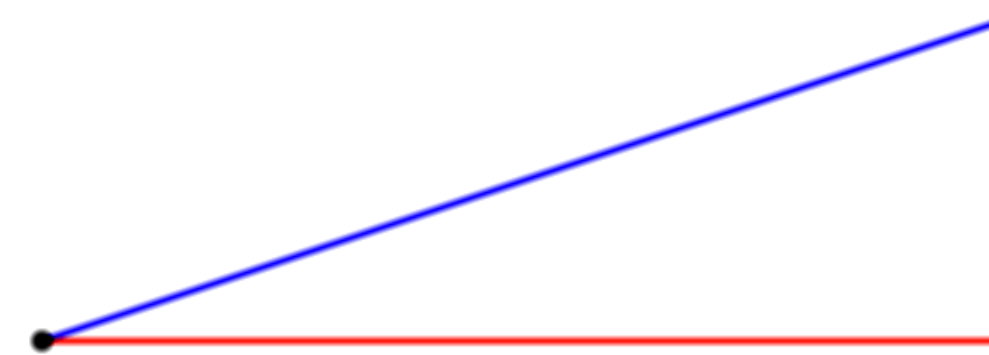
How far apart are they?



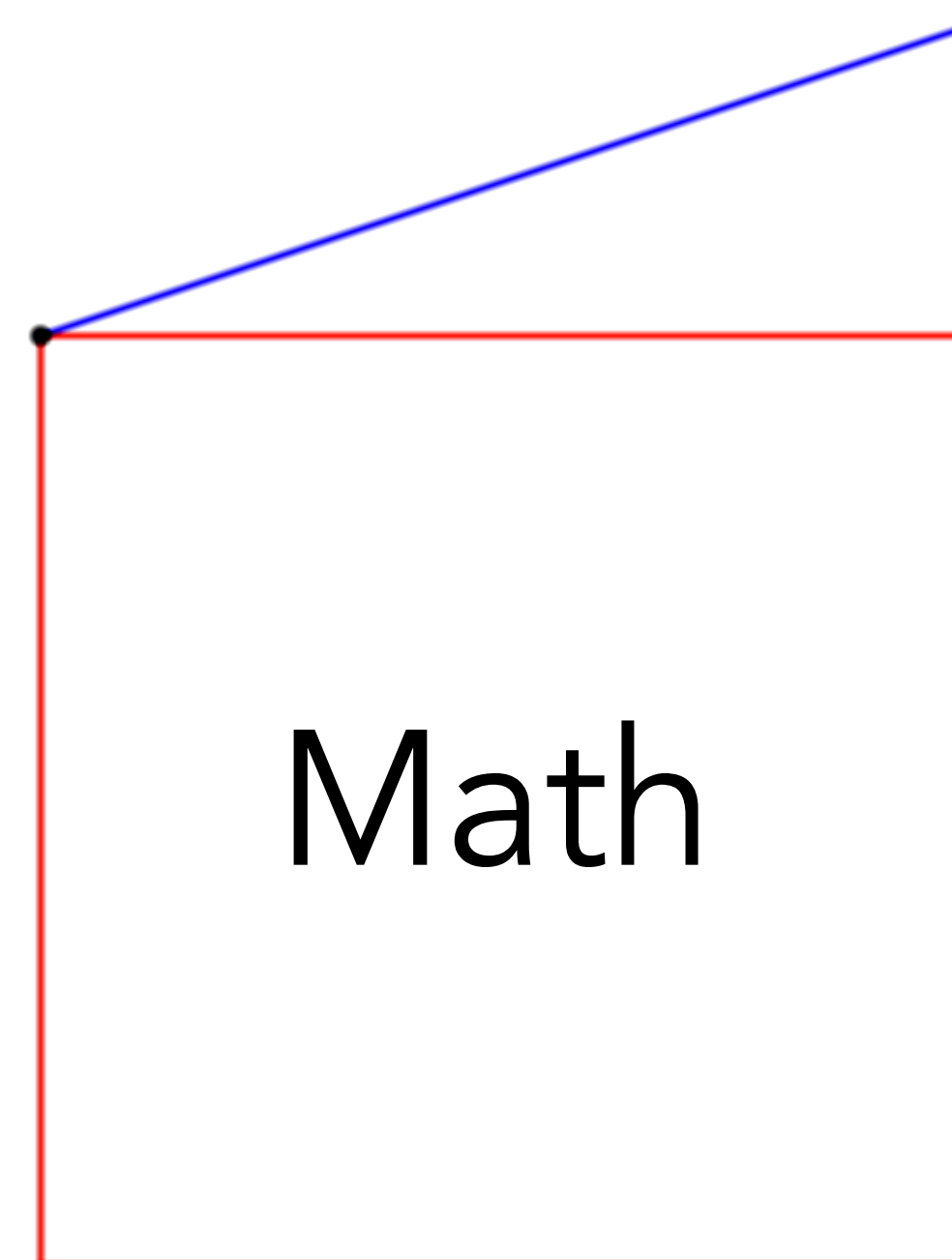
Time for math.

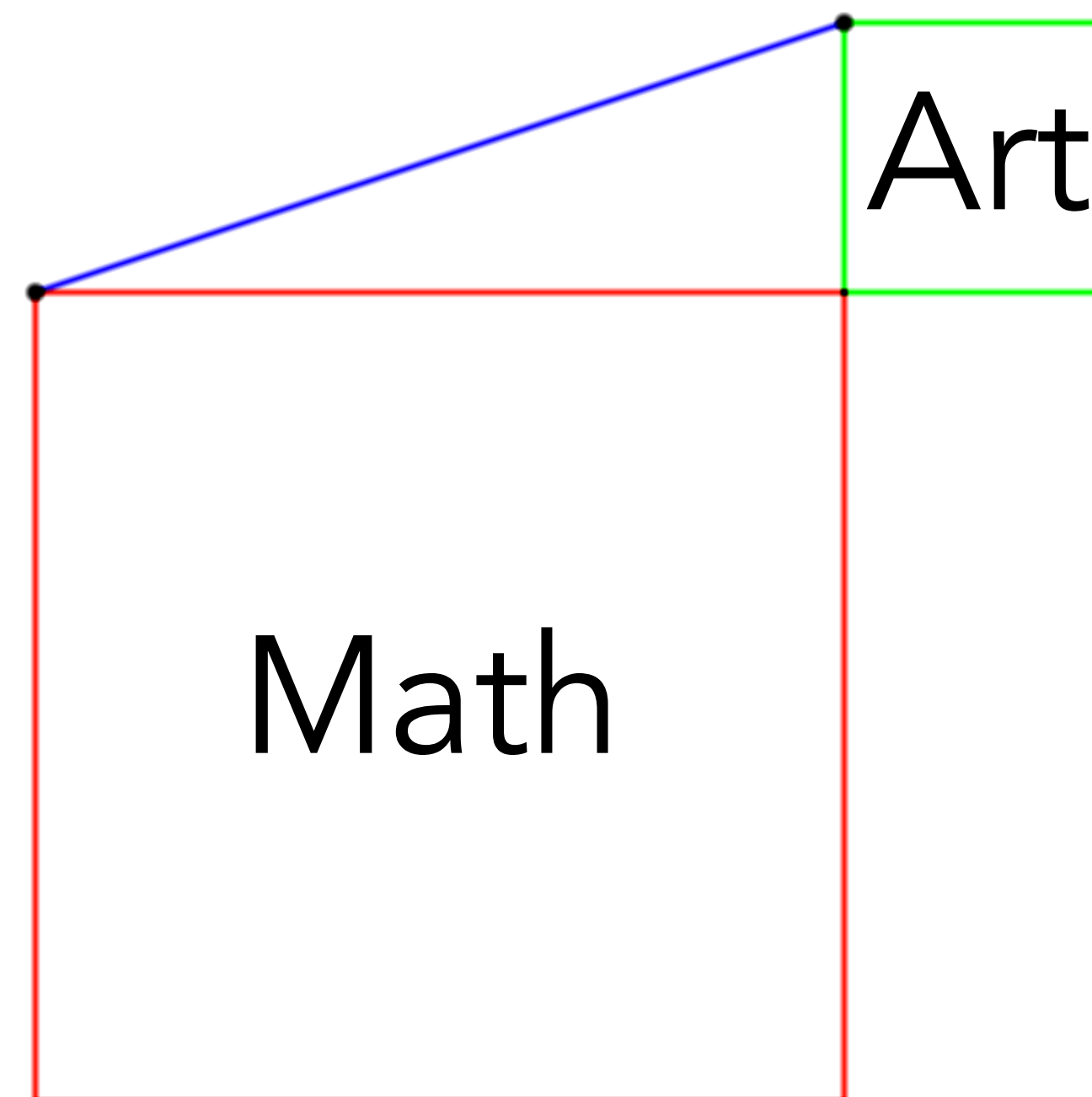


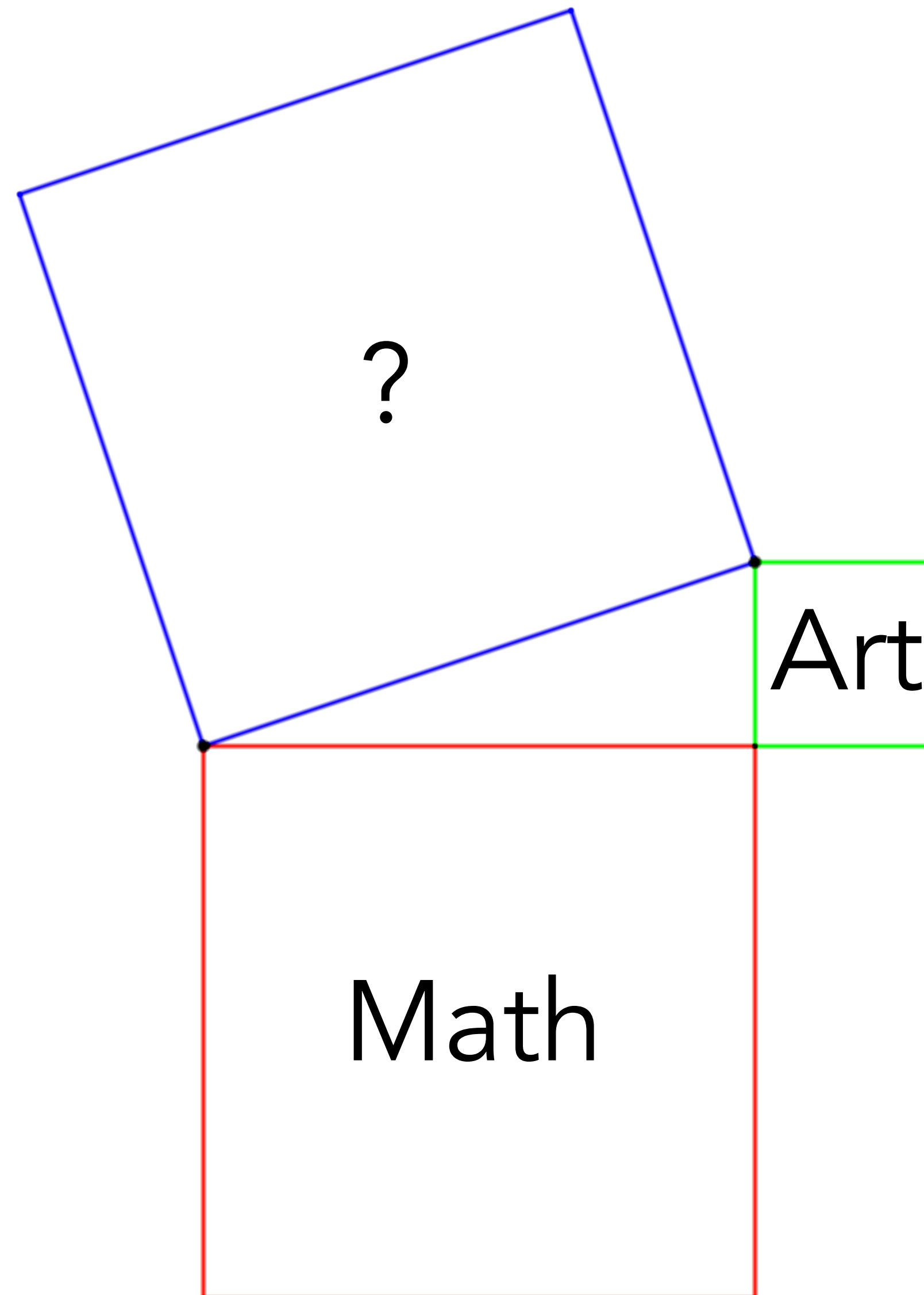
But it's hard to measure math against art.



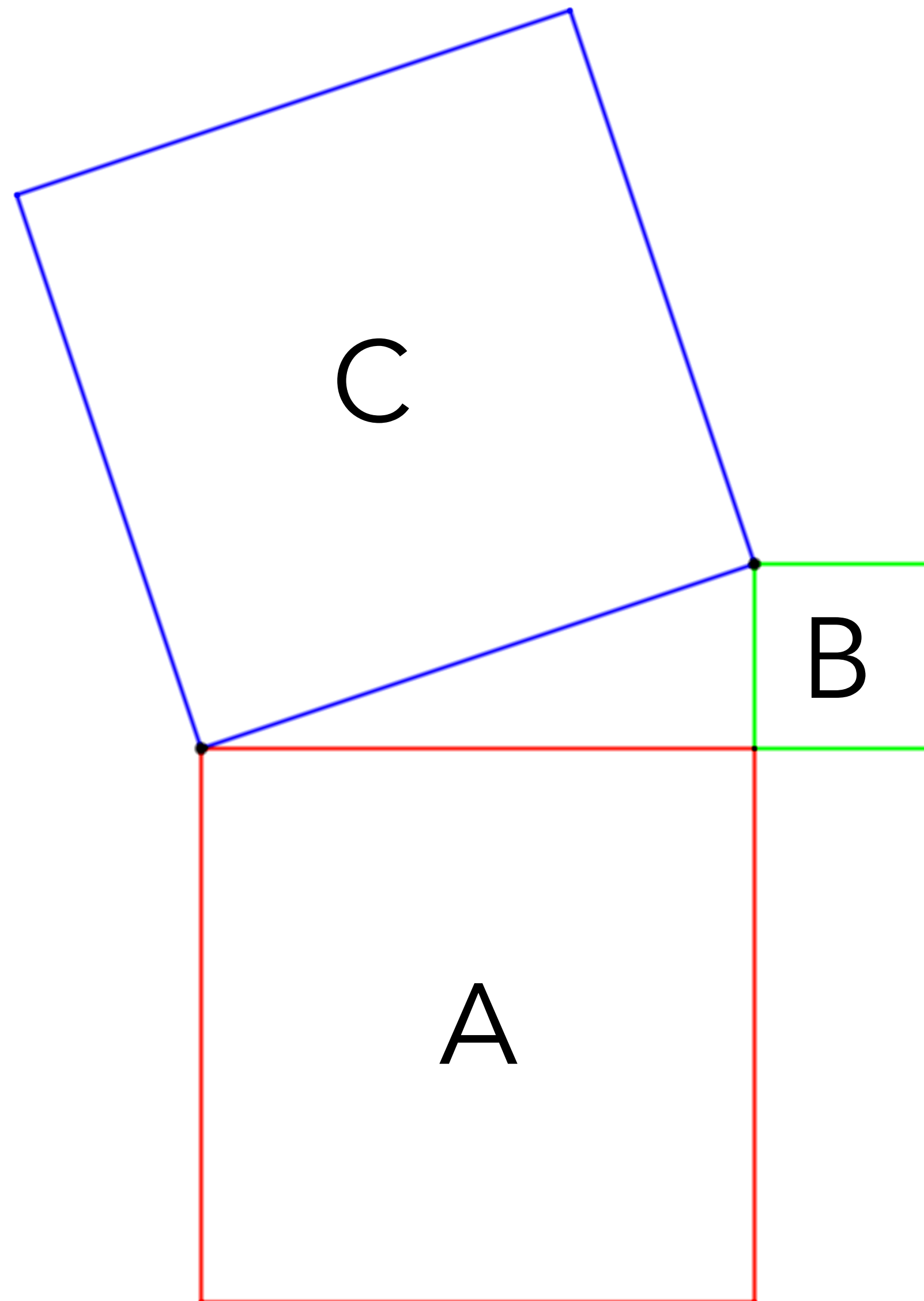
That blue line isn't one or the other.



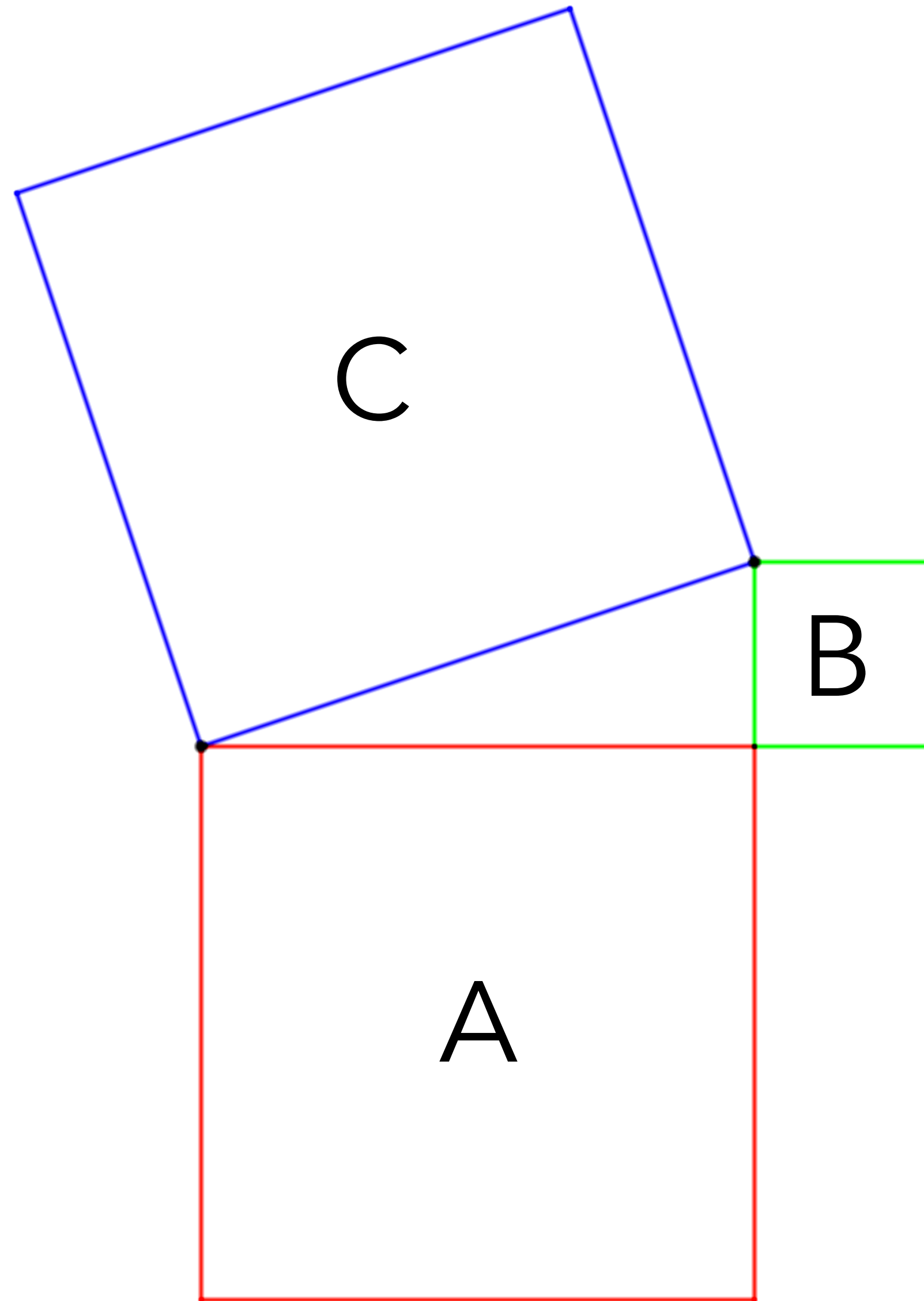




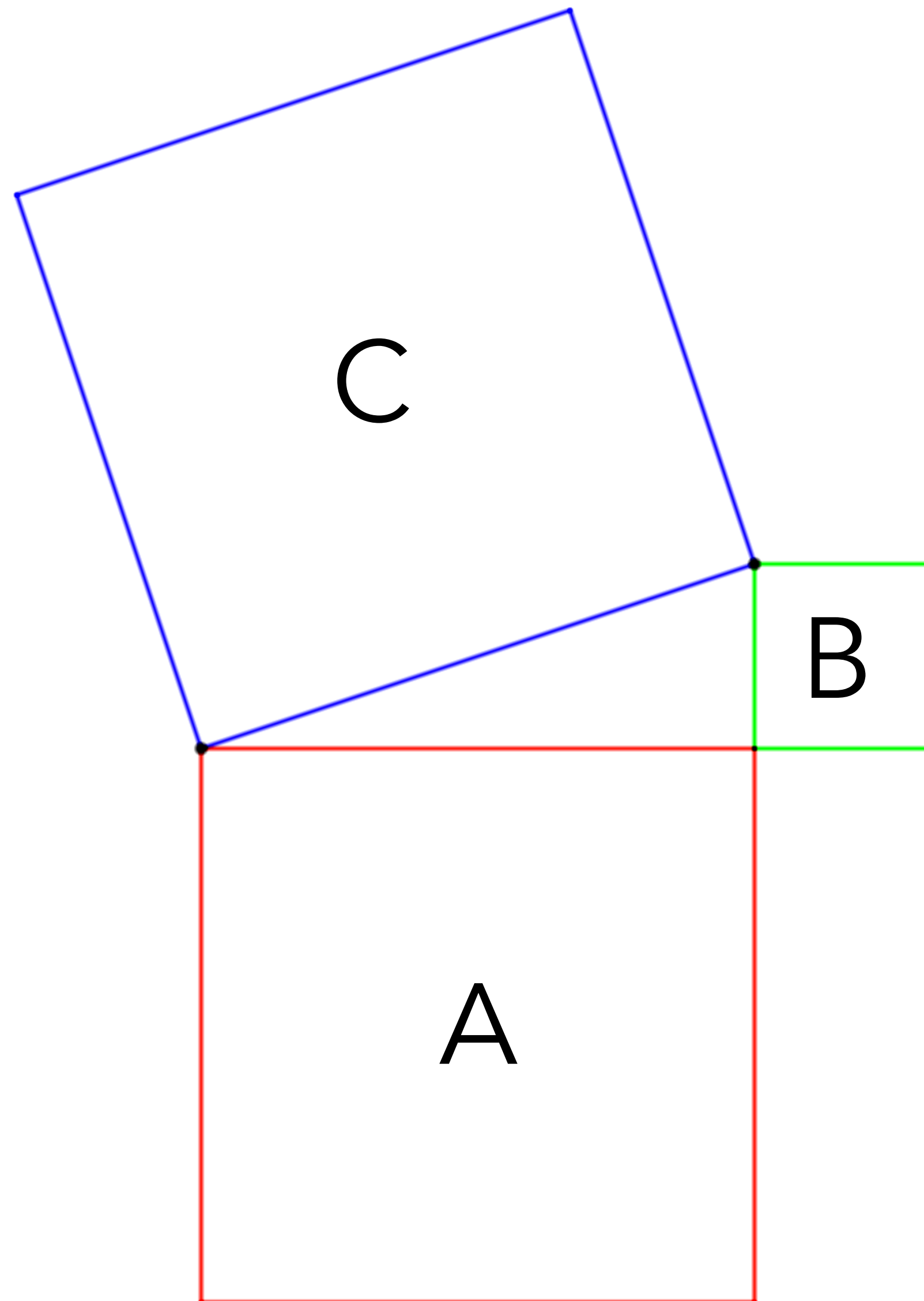
$$A^*A + B^*B = C^*C$$

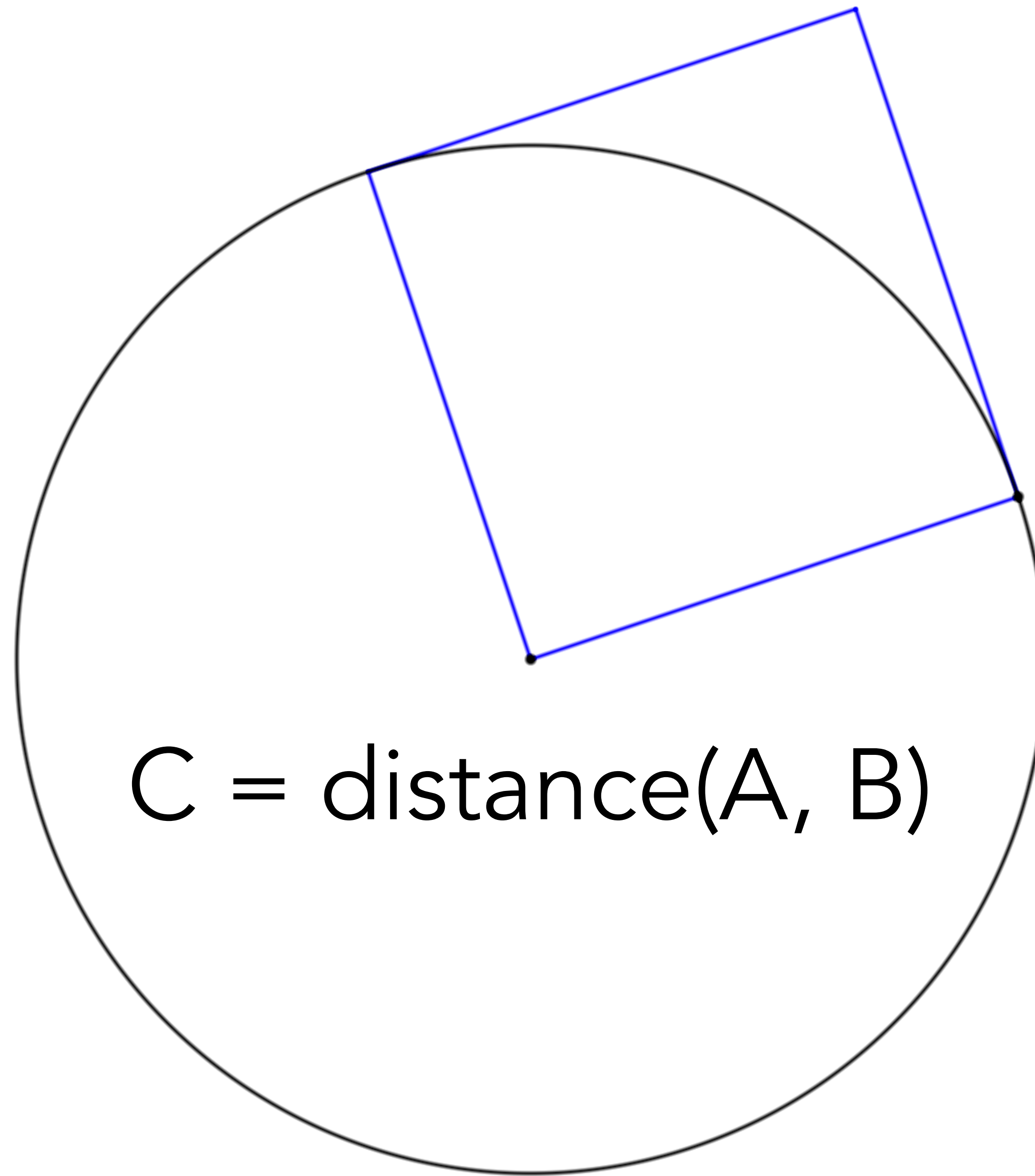


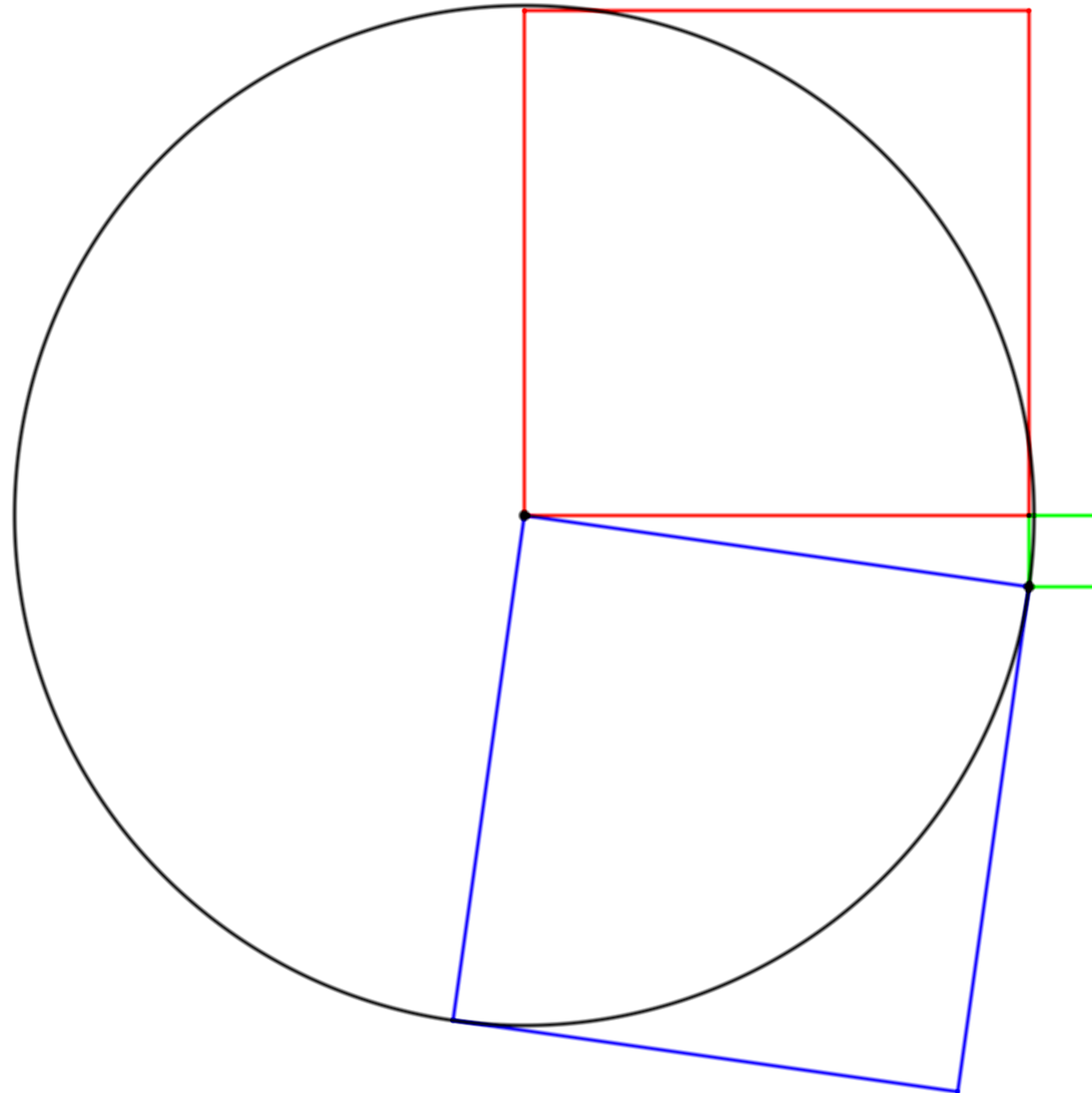
$$\text{sqrt}(A * A + B * B) = C$$



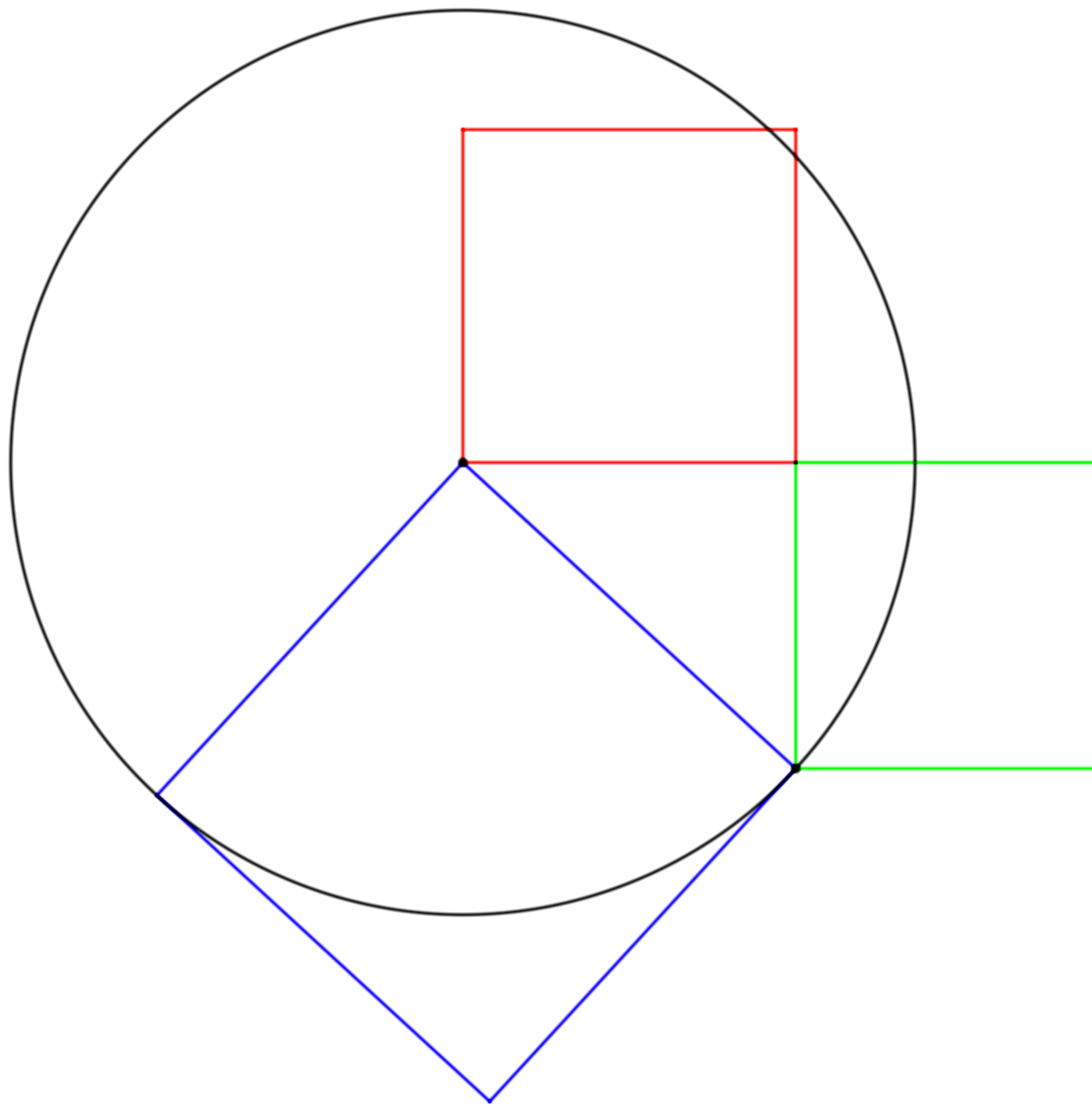
$$C = \text{distance}(A, B)$$

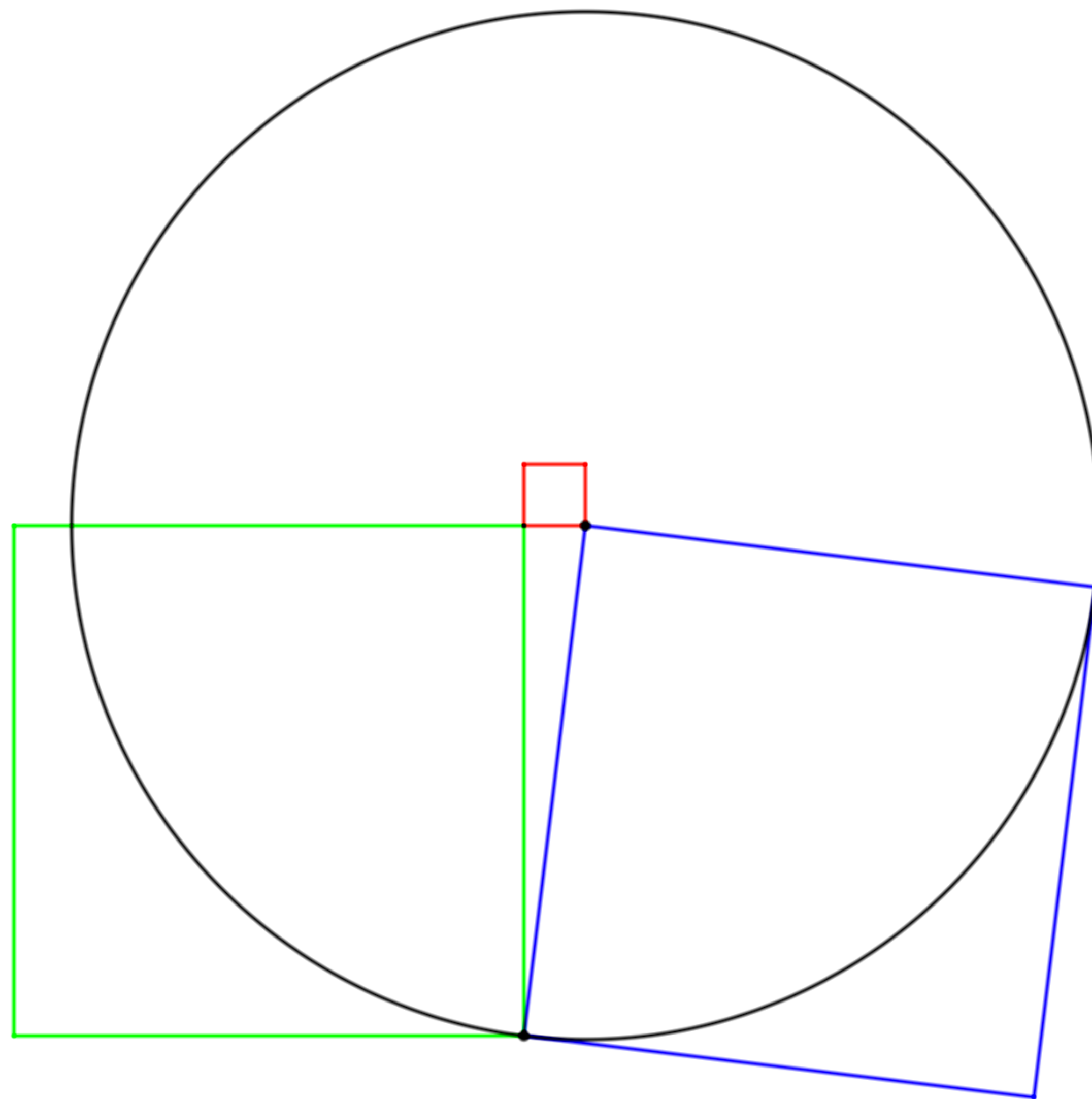


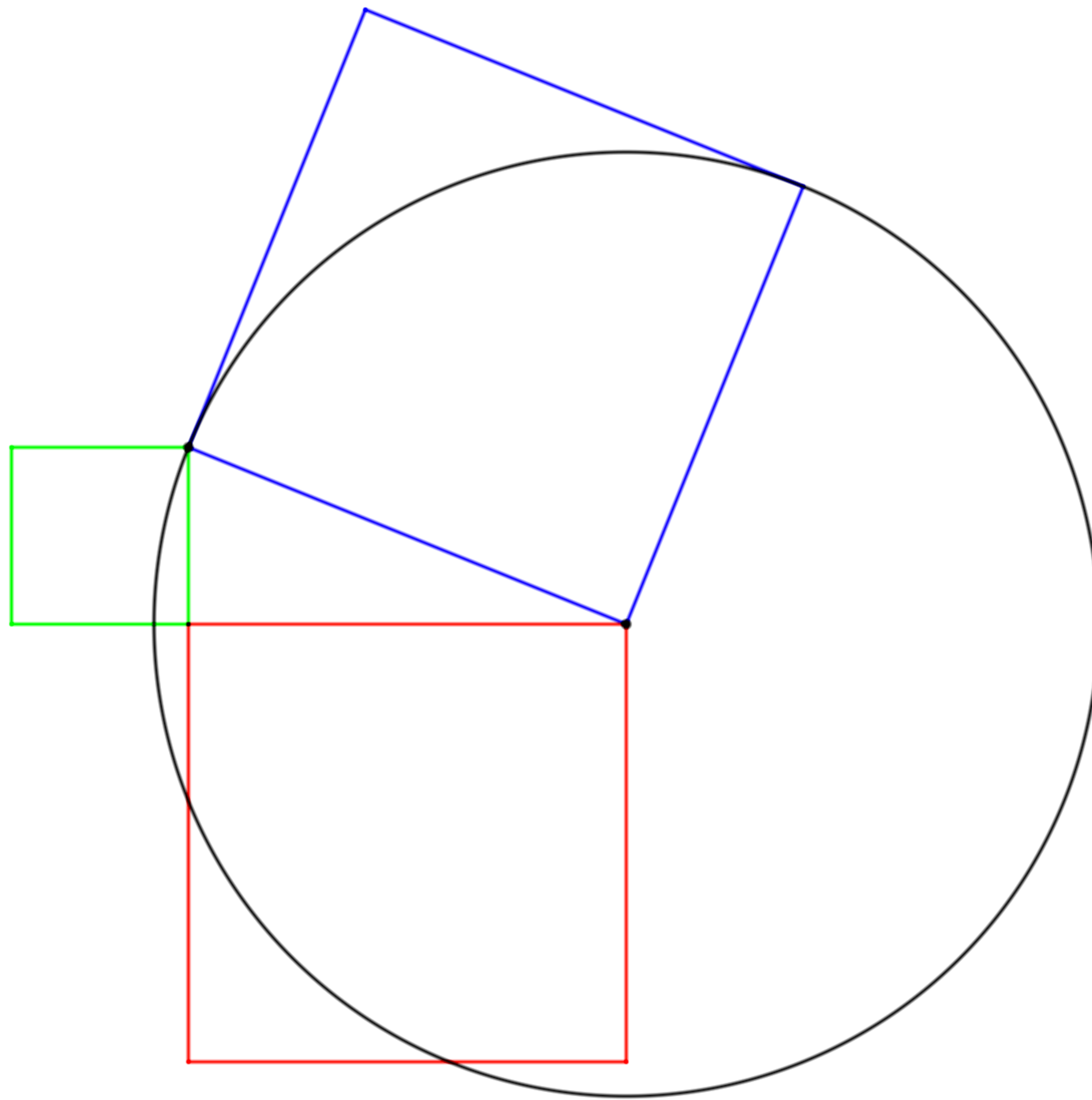


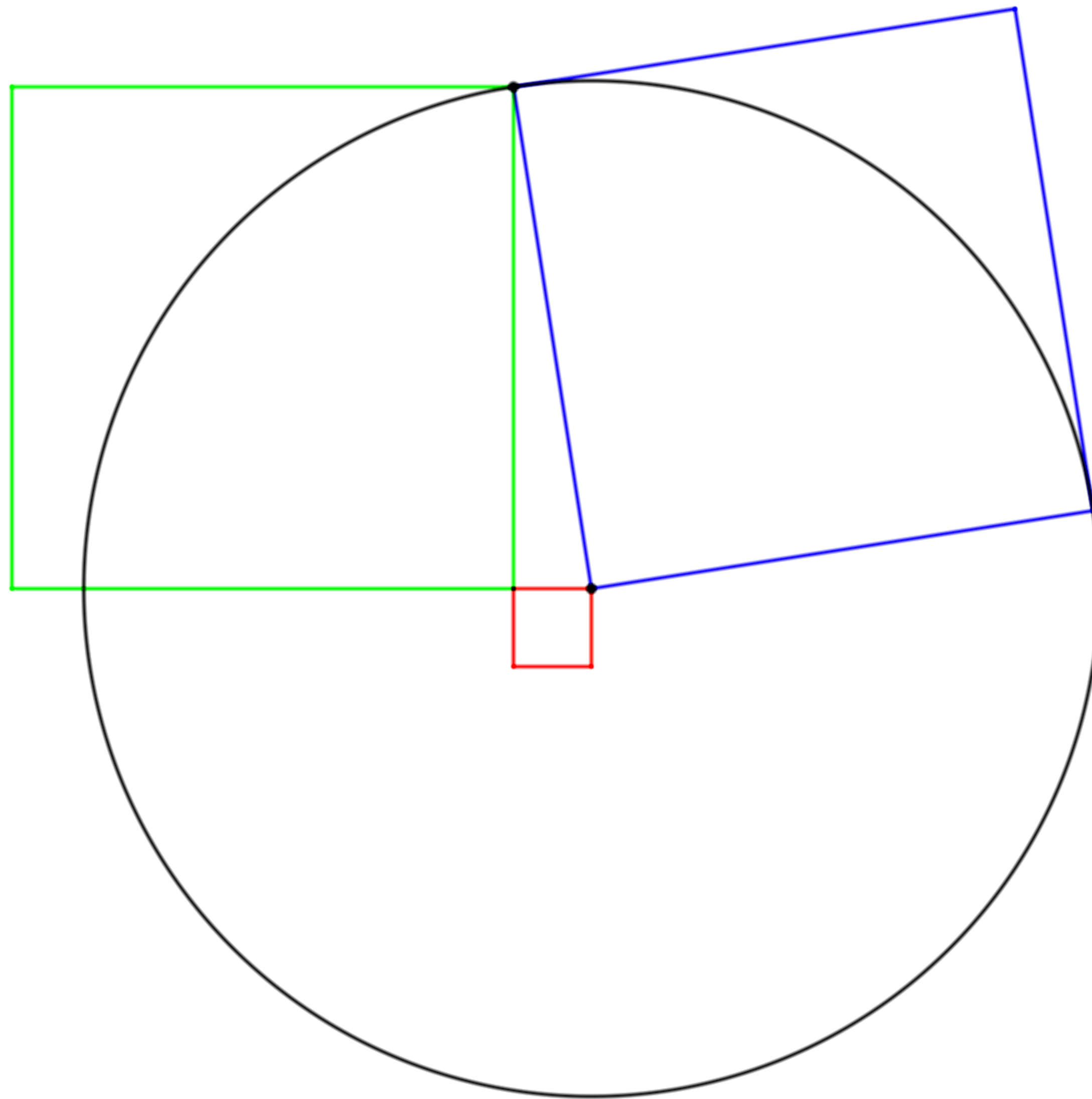


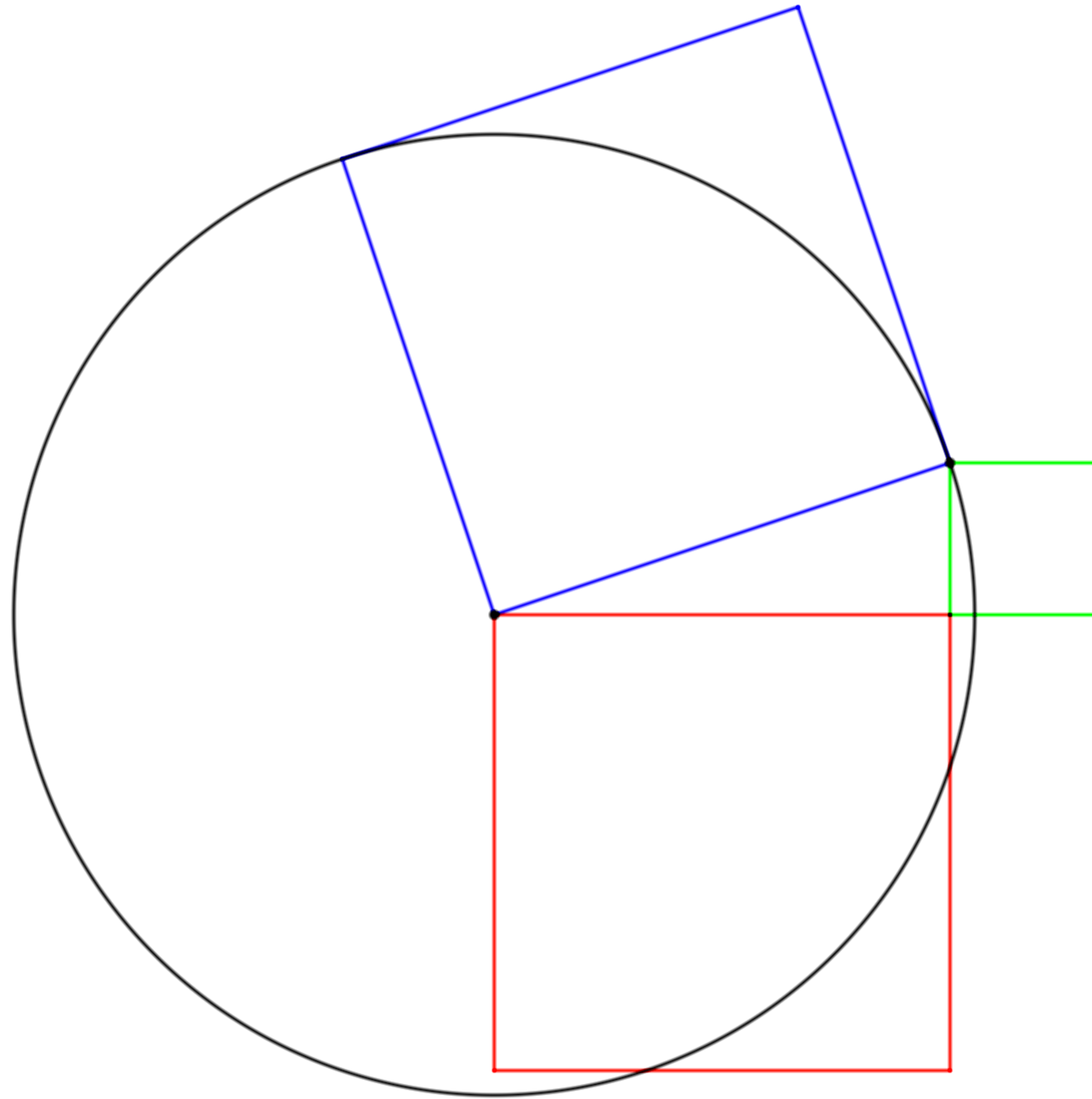
This circle is the same distance in every direction.



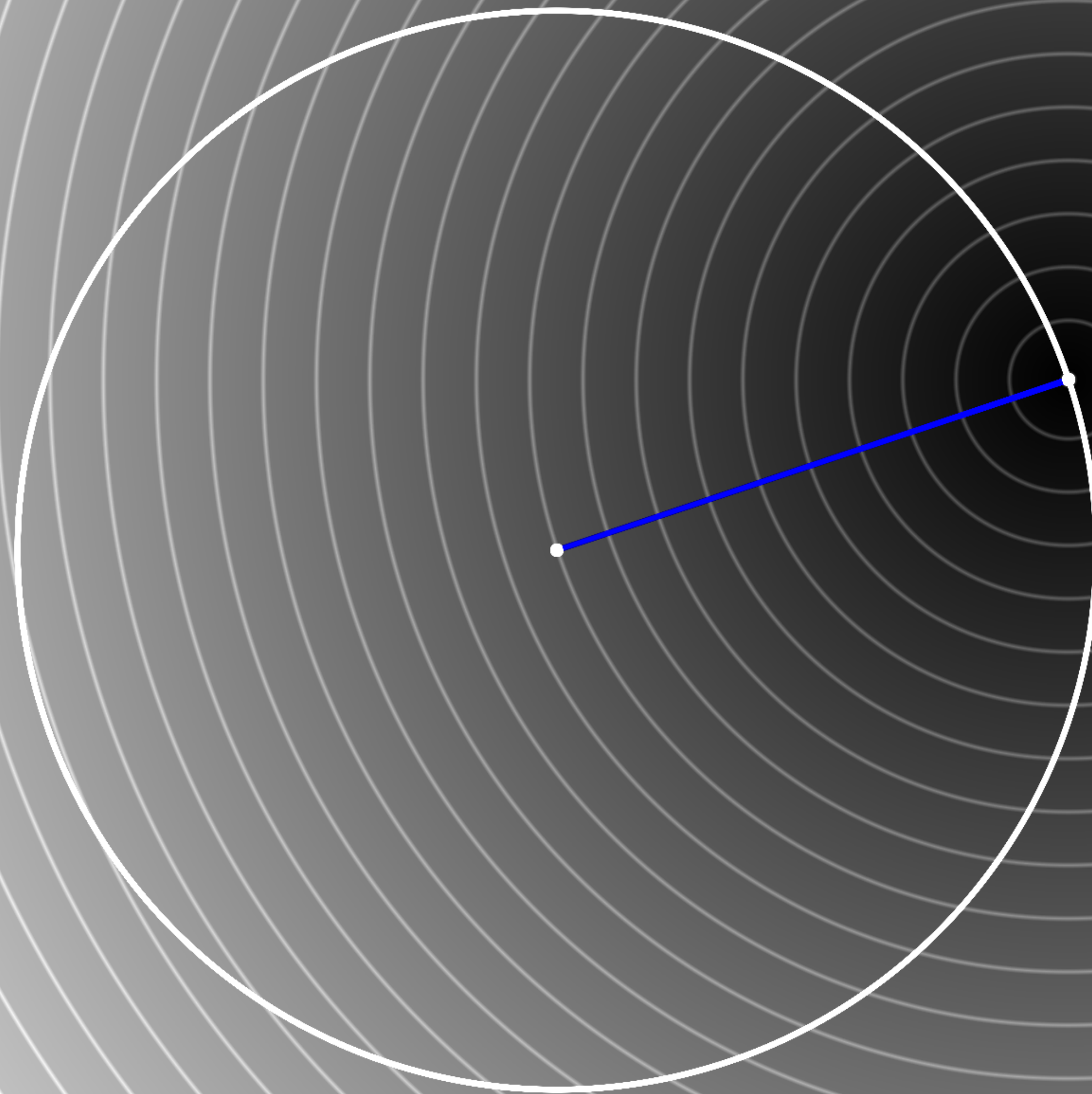




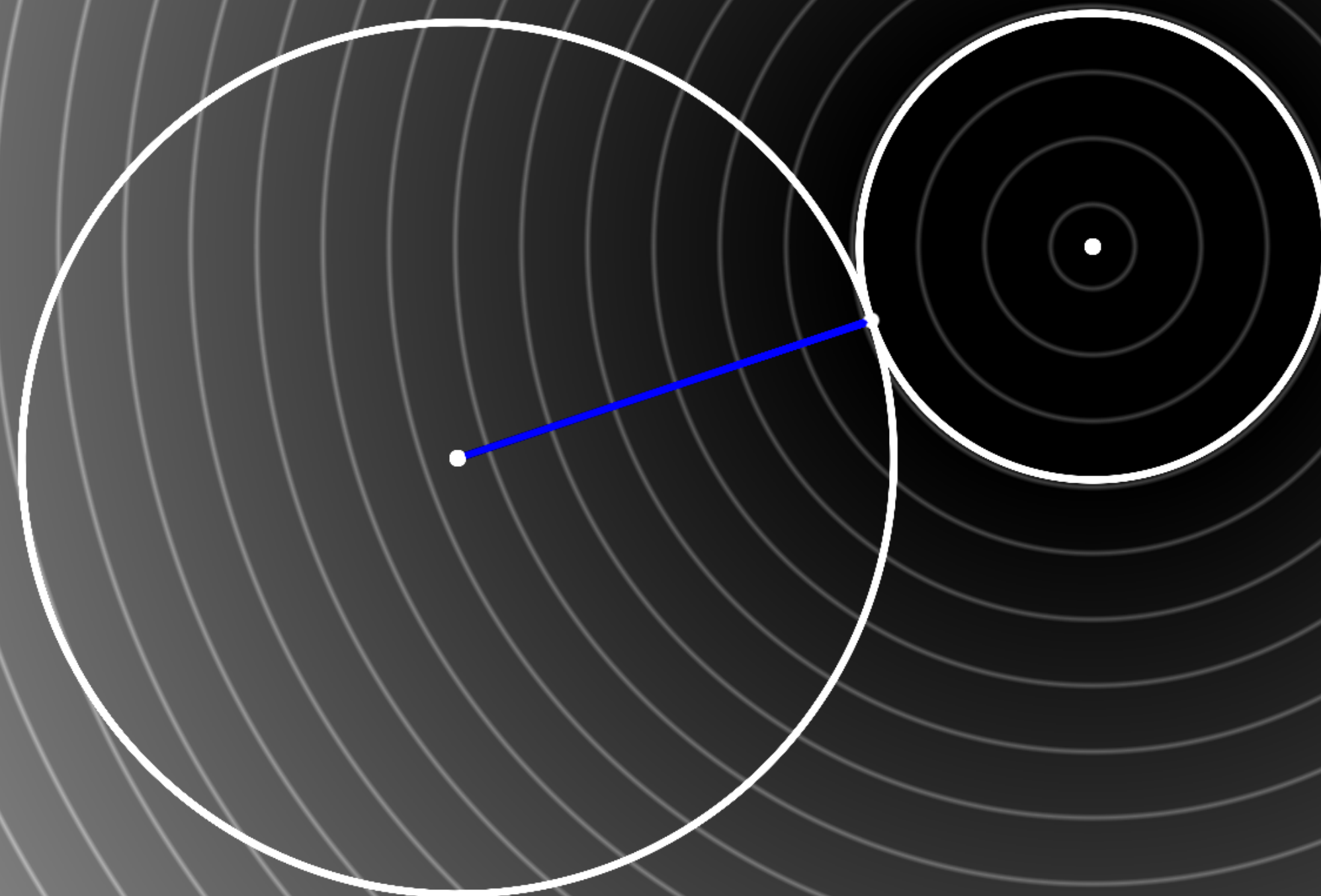




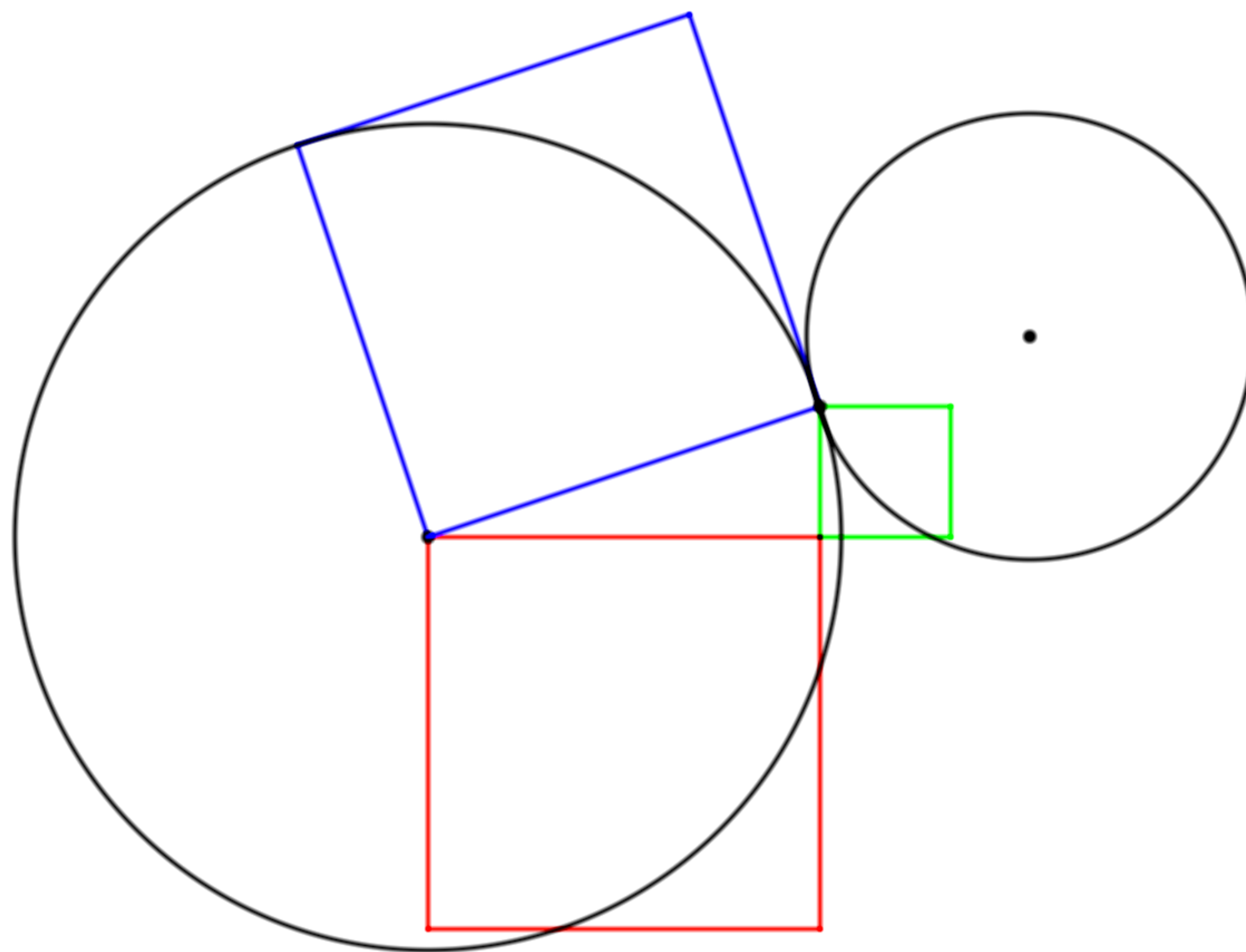
Which means this is only the distance, not the angle.

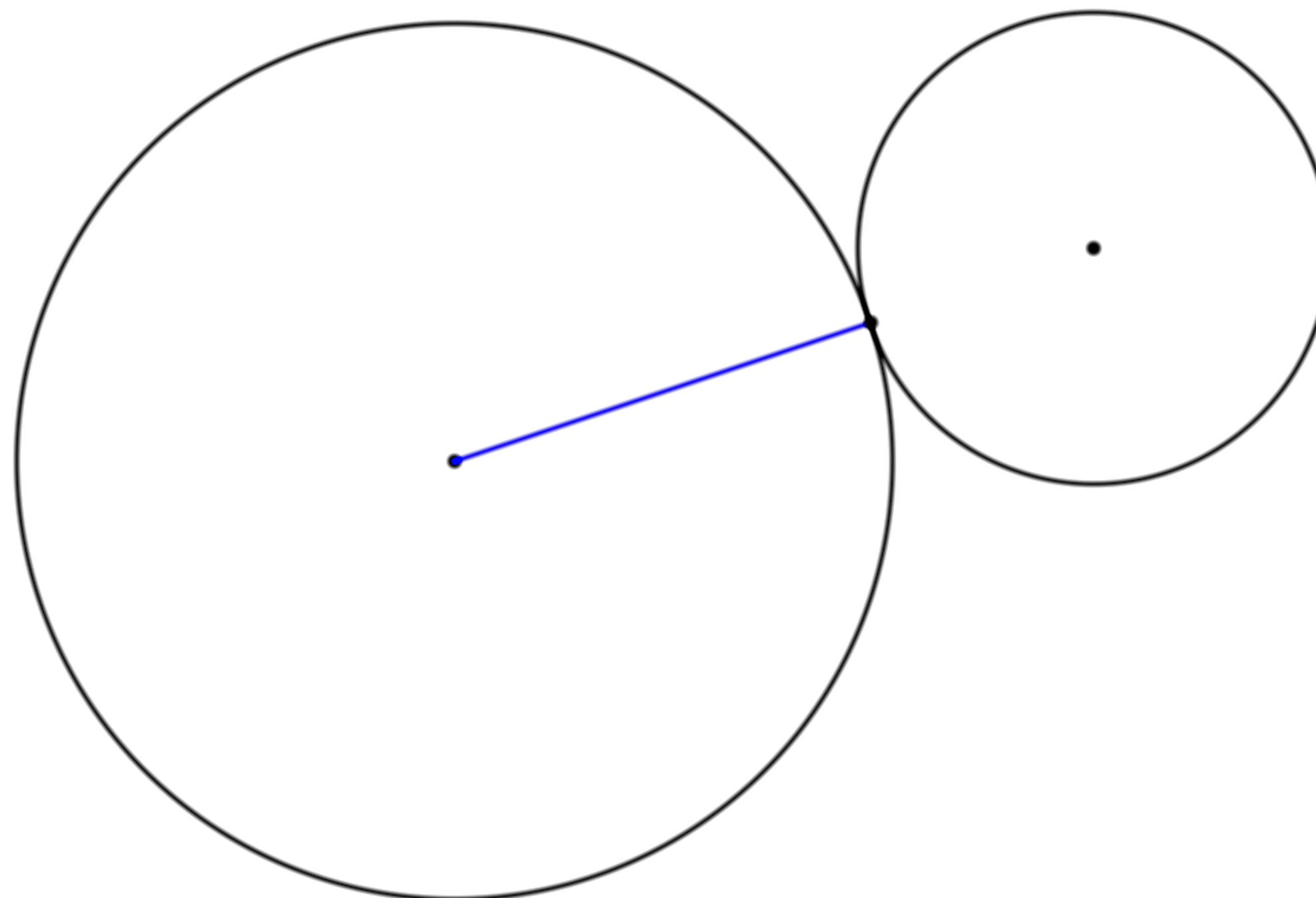


The distance field away from the second point.



Subtract a radius, and the distance field is now a circle.







Sphere Tracing



Sphere Tracing

Sphere tracing is a form of raytracing
(there are many.)

Sphere Tracing

Sphere tracing is a form of raytracing
(there are many.)

It is ideal for implicit functions.

Sphere Tracing

Sphere tracing is a form of raytracing
(there are many.)

It is ideal for implicit functions.

It is not a replacement for rasterization or voxels.



Sphere Tracing

It is inefficient.



Sphere Tracing

It is inefficient.

But, it is simple.



Sphere Tracing

It is inefficient.

But, it is simple.

And it is extremely flexible.



Sphere Tracing

4 Steps



Construct the view.

Sphere Tracing

4 Steps



Sphere Tracing

4 Steps

Construct the view.

Trace the ray.



Sphere Tracing

4 Steps

Construct the view.

Trace the ray.

Determine which way the surface is facing.



Sphere Tracing

4 Steps

Construct the view.

Trace the ray.

Determine which way the surface is facing.

Add light.

1.

Constructing the view.

Two triangles.

UV coordinates.



That's all you need.



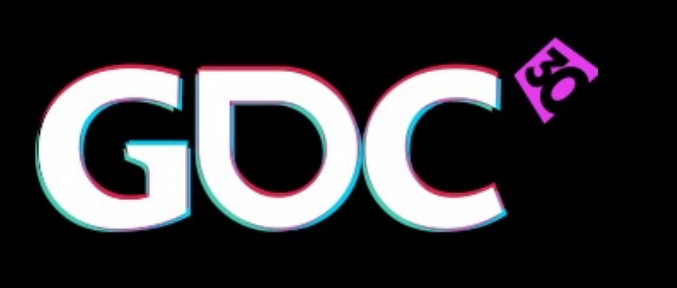
Now let's code!


```
vec2 screen_coordinates = gl_FragCoord.xy;  
screen_coordinates /= resolution;  
screen_coordinates = screen_coordinates - .5;  
screen_coordinates *= resolution/min(resolution.x, resolution.y);  
  
float field_of_view = 1.5;  
  
vec3 direction = vec3(screen_coordinates, field_of_view);  
direction = normalize(direction);
```

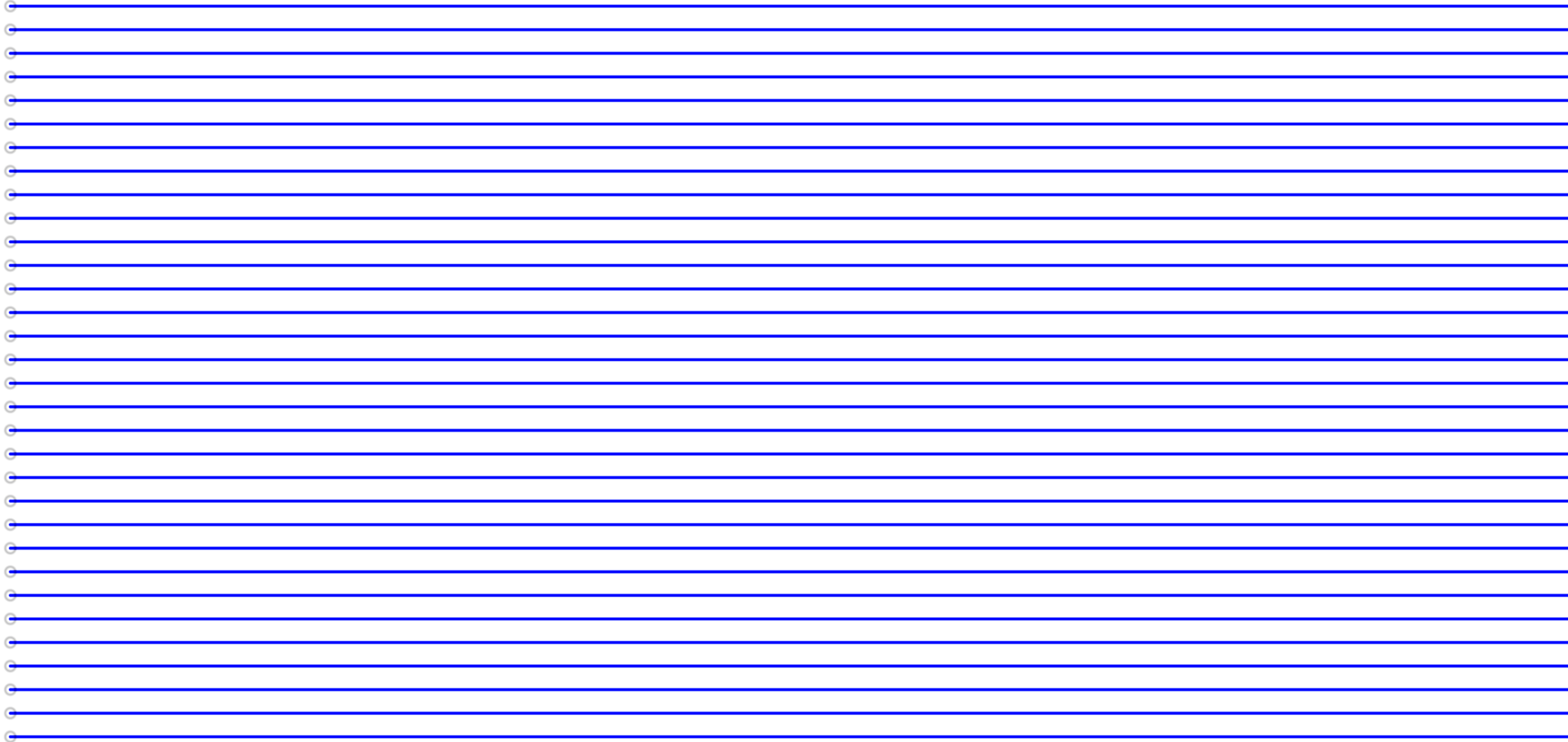
```
vec2 screen_coordinates = gl_FragCoord.xy;  
screen_coordinates /= resolution;  
screen_coordinates = screen_coordinates - .5;  
screen_coordinates *= resolution/min(resolution.x, resolution.y);  
  
float field_of_view = 1.5;  
  
vec3 direction = vec3(screen_coordinates, field_of_view);  
direction = normalize(direction);
```



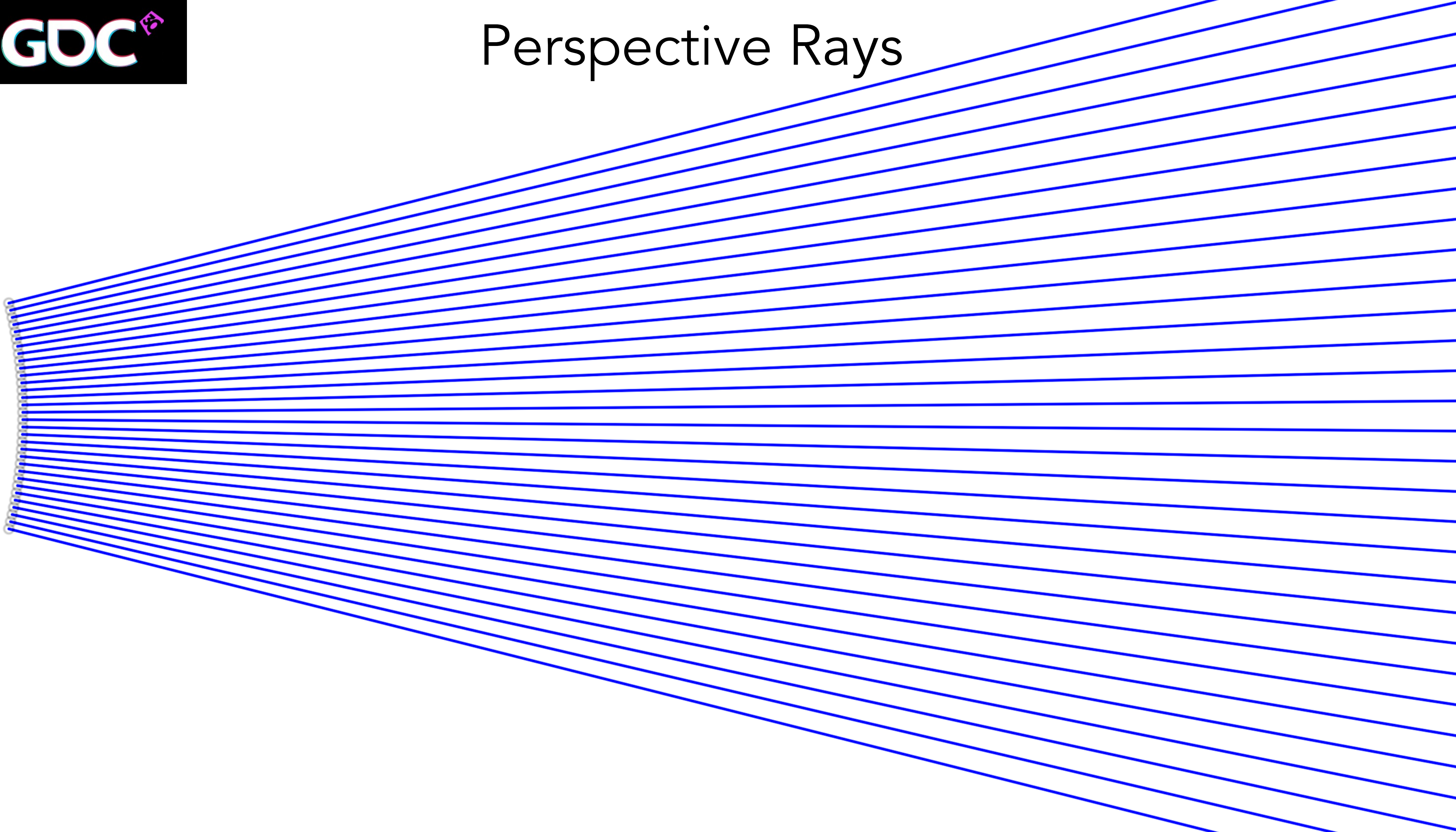
```
vec2 screen_coordinates = gl_FragCoord.xy;  
screen_coordinates /= resolution;  
screen_coordinates = screen_coordinates - .5;  
screen_coordinates *= resolution/min(resolution.x, resolution.y);  
  
float field_of_view = 1.5;  
  
vec3 direction = vec3(screen_coordinates, field_of_view);  
direction = normalize(direction);
```



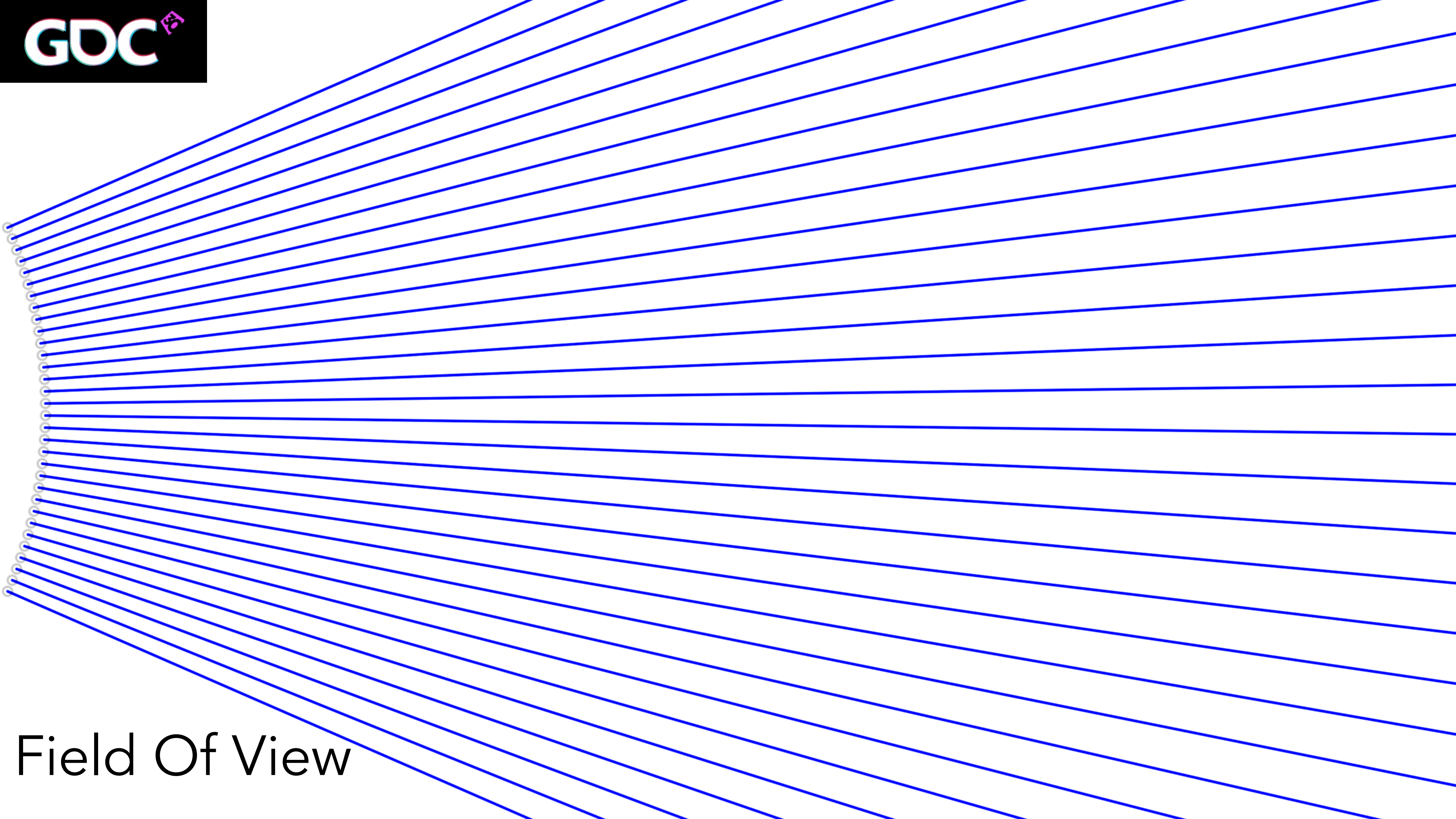
Orthogonal Rays



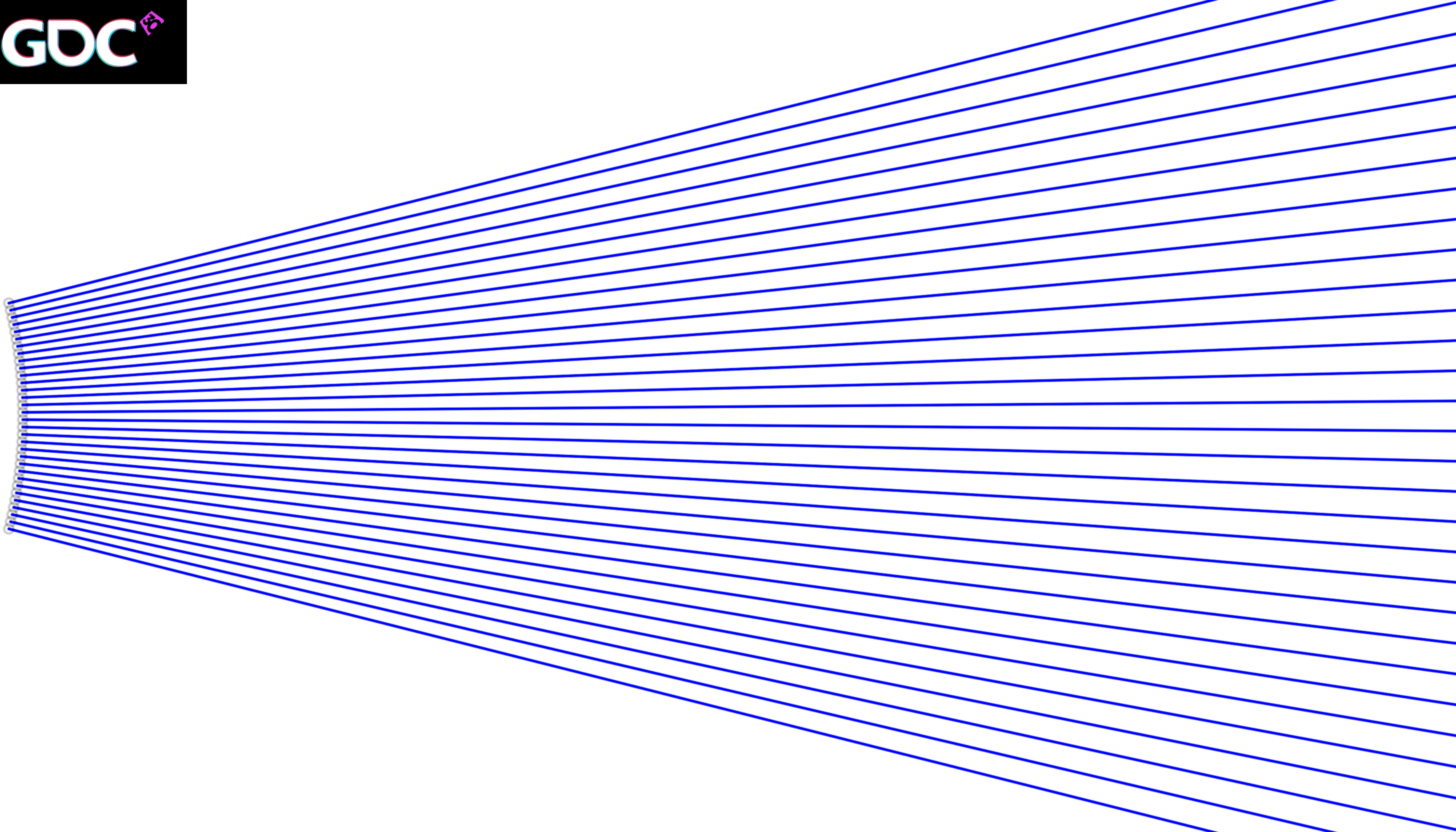
Perspective Rays

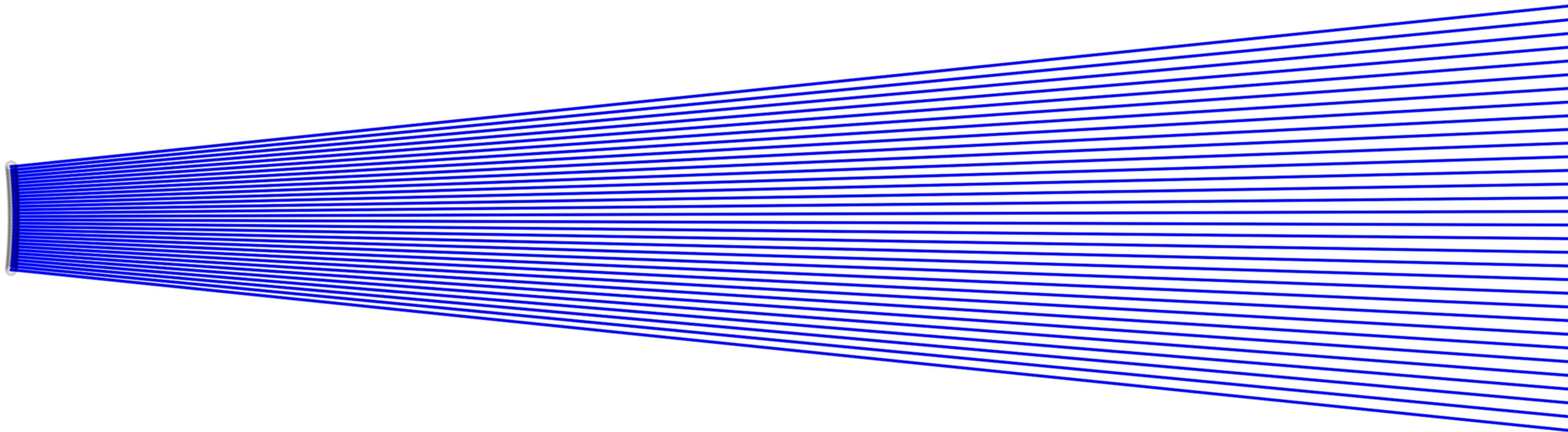


```
vec2 screen_coordinates = gl_FragCoord.xy;  
screen_coordinates /= resolution;  
screen_coordinates = screen_coordinates - .5;  
screen_coordinates *= resolution/min(resolution.x, resolution.y);  
  
float field_of_view = 1.5;  
  
vec3 direction = vec3(screen_coordinates, field_of_view);  
direction = normalize(direction);
```

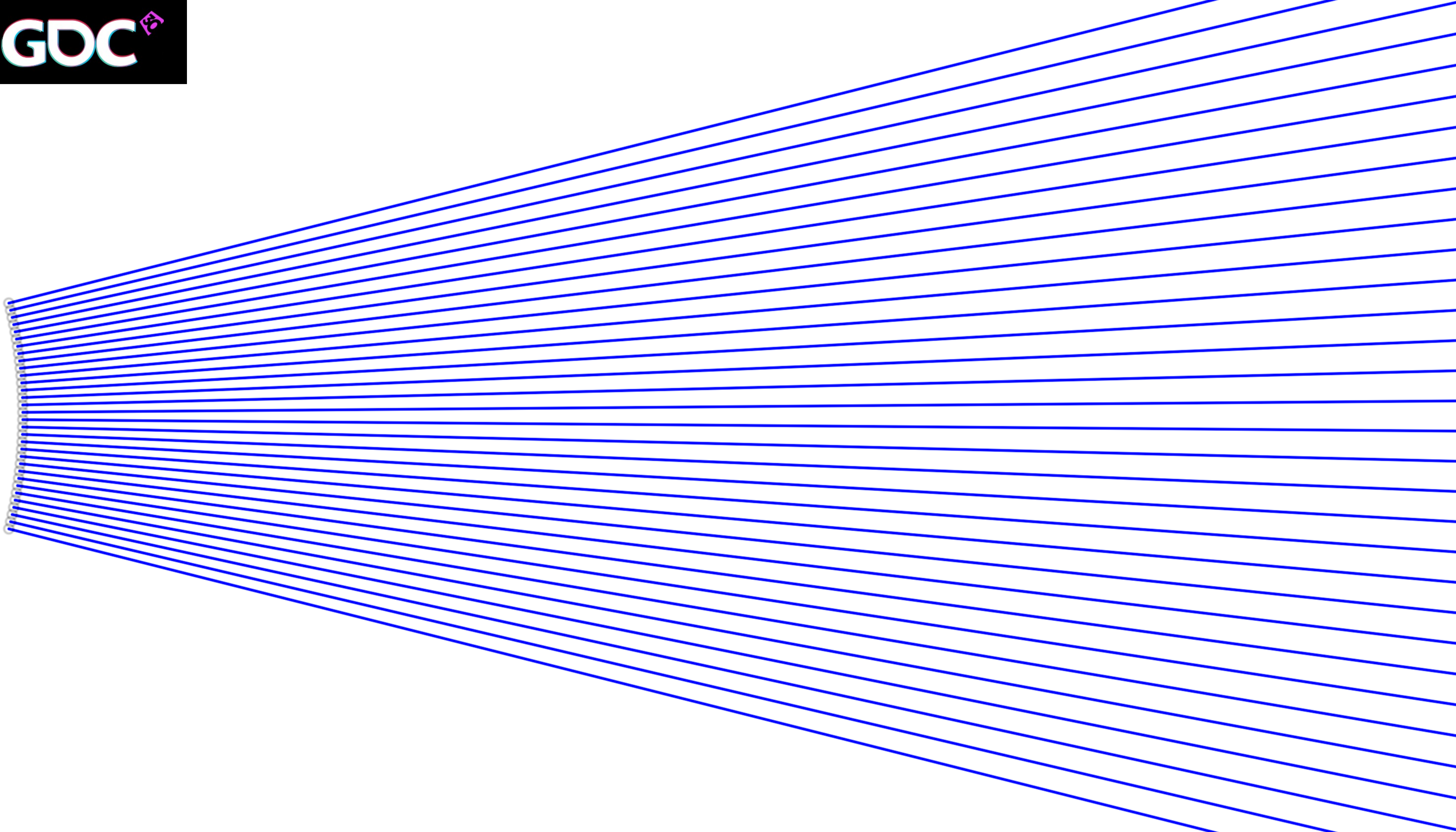



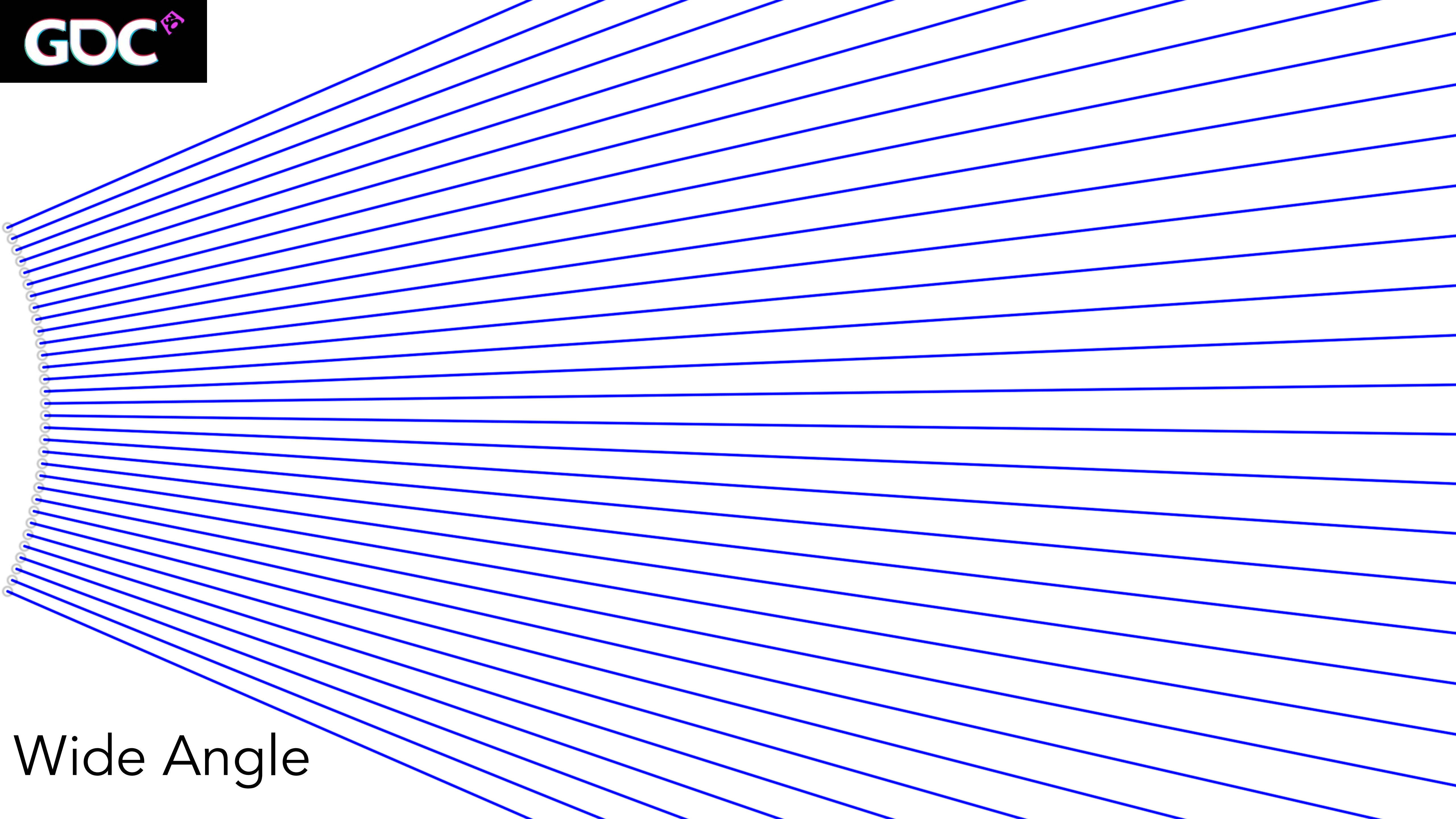
Field Of View





Zoom



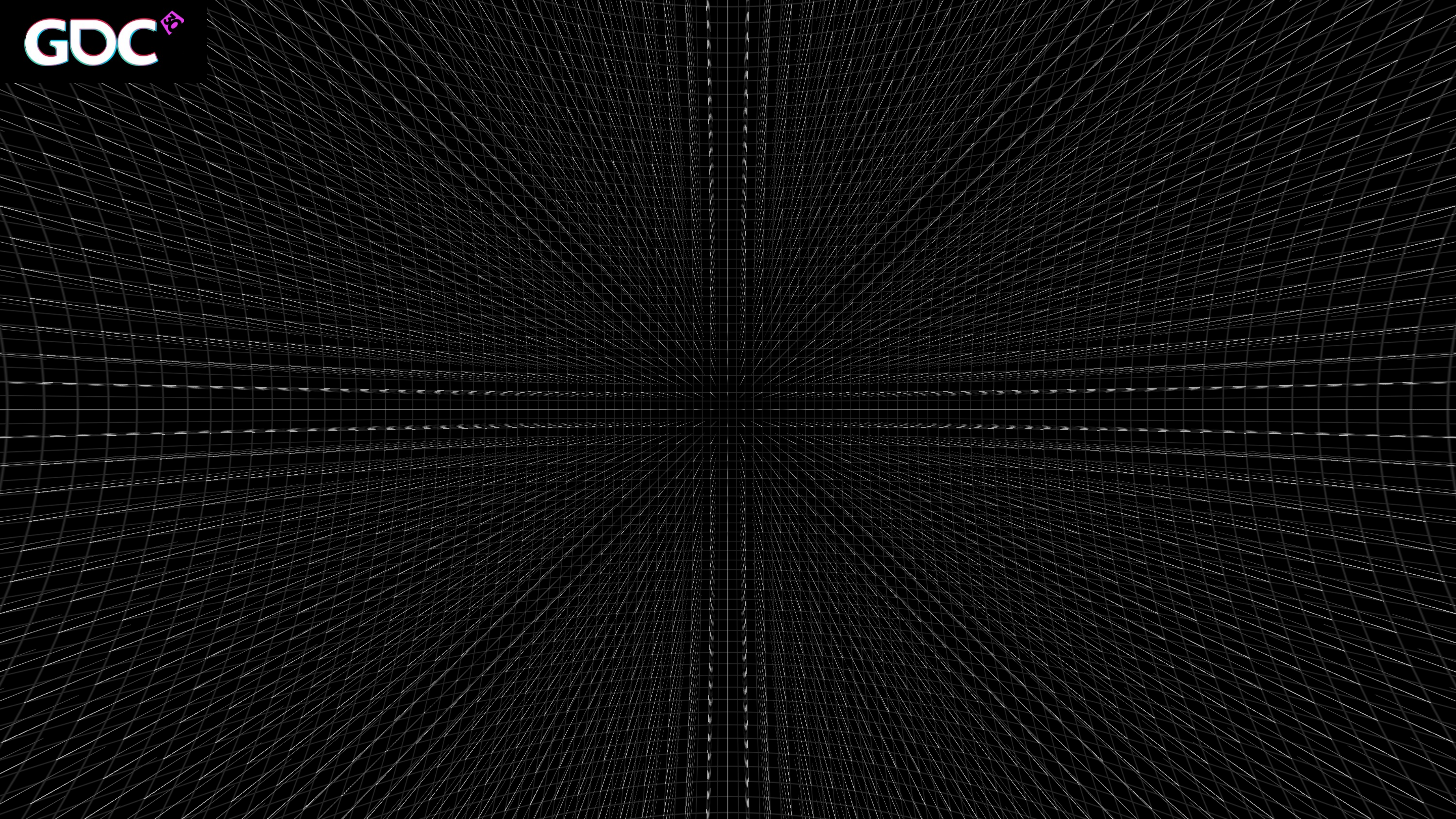


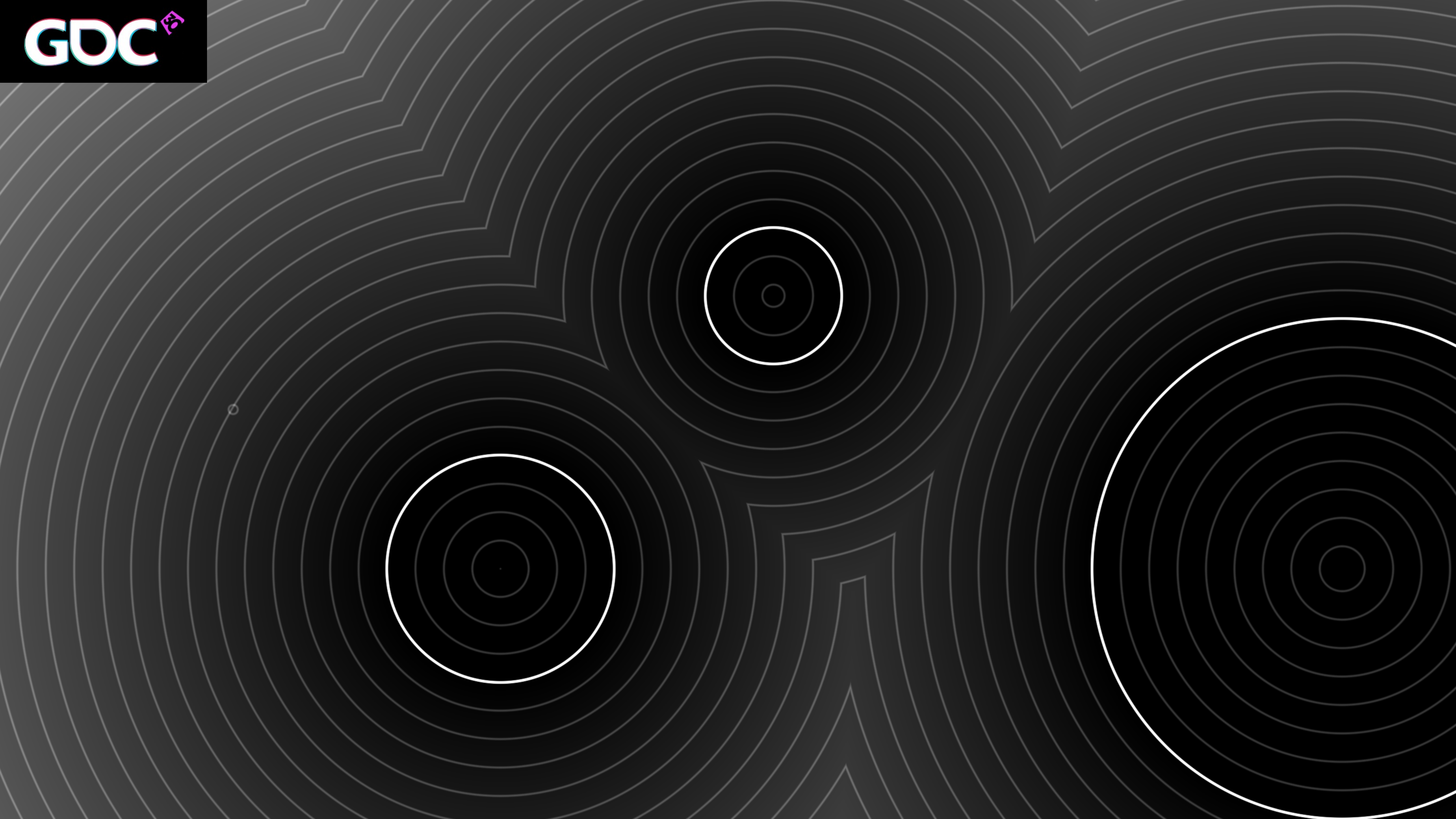
Wide Angle



(A jump to 3D for a moment.)


```
vec2 screen_coordinates = gl_FragCoord.xy;  
screen_coordinates /= resolution;  
screen_coordinates = screen_coordinates - .5;  
screen_coordinates *= resolution/min(resolution.x, resolution.y);  
  
float field_of_view = 1.5;  
  
vec3 direction = vec3(screen_coordinates, field_of_view);  
direction = normalize(direction);
```



2.



Trace the ray.


```
vec2 origin = vec2(0.);
```



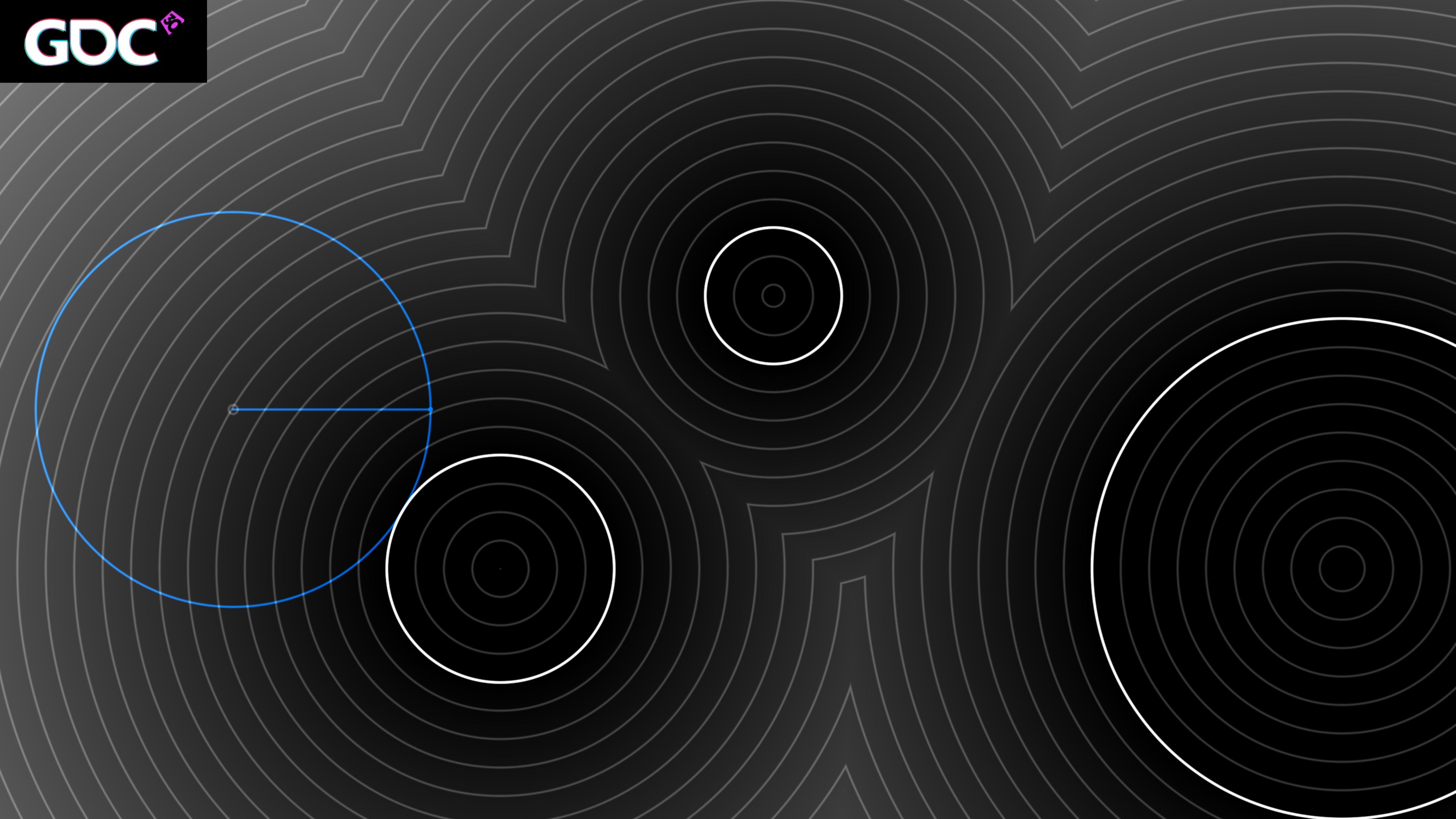

```
vec2 origin          = vec2(0.);  
vec2 position        = origin;  
  
float surface_threshold = .001;  
for(int i = 0; i < 128; i++)  
{  
    float distance_to_surface = map(position);  
    if(distance_to_surface < surface_threshold) break;  
    position += direction * distance_to_surface;  
}  
  
float distance_to_scene = distance(origin, position);
```



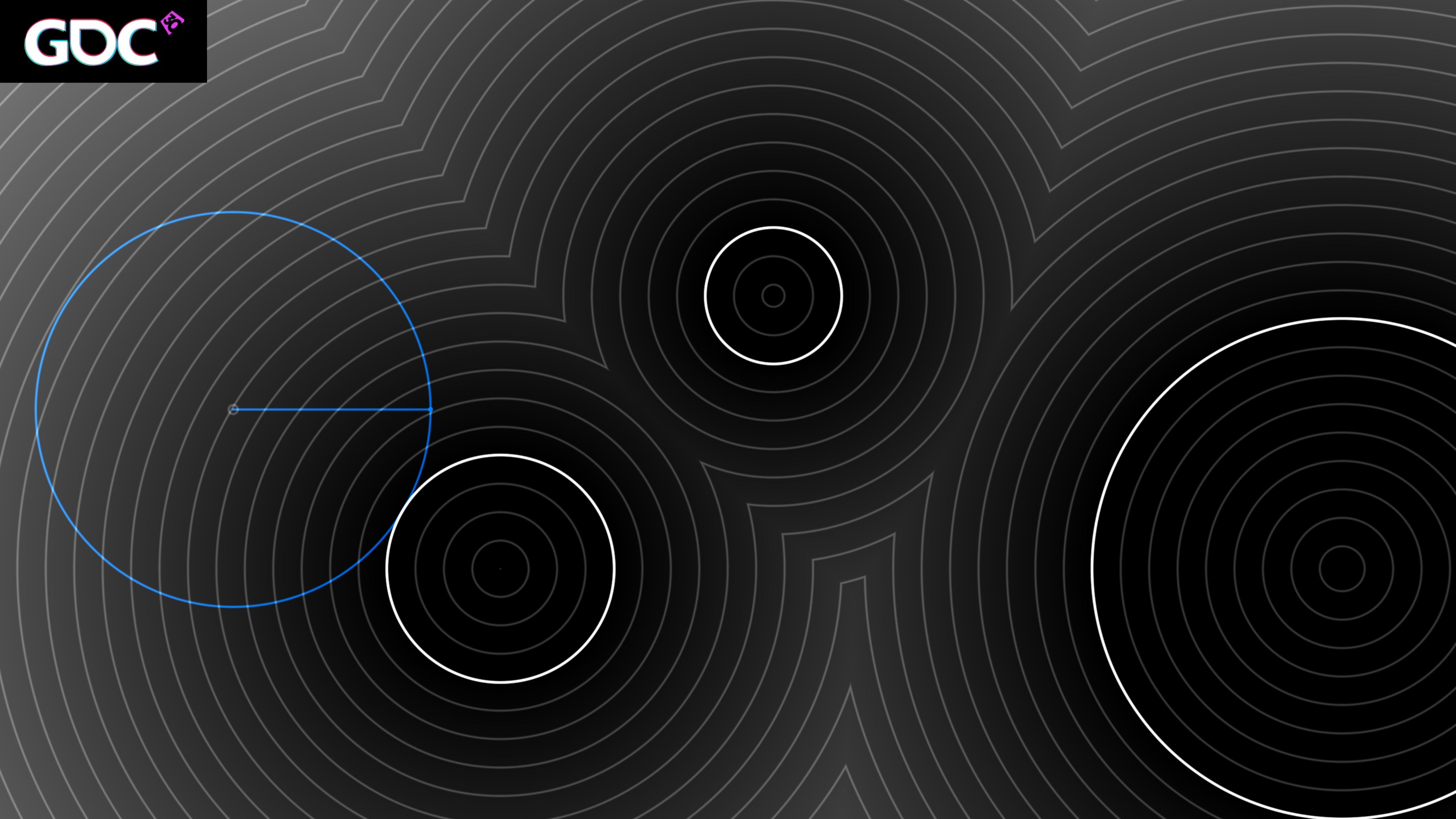
```
vec2 origin          = vec2(0.);  
vec2 position        = origin;  
  
float surface_threshold = .001;  
for(int i = 0; i < 128; i++)  
{  
    float distance_to_surface = map(position);  
    if(distance_to_surface < surface_threshold) break;  
    position += direction * distance_to_surface;  
}  
  
float distance_to_scene = distance(origin, position);
```



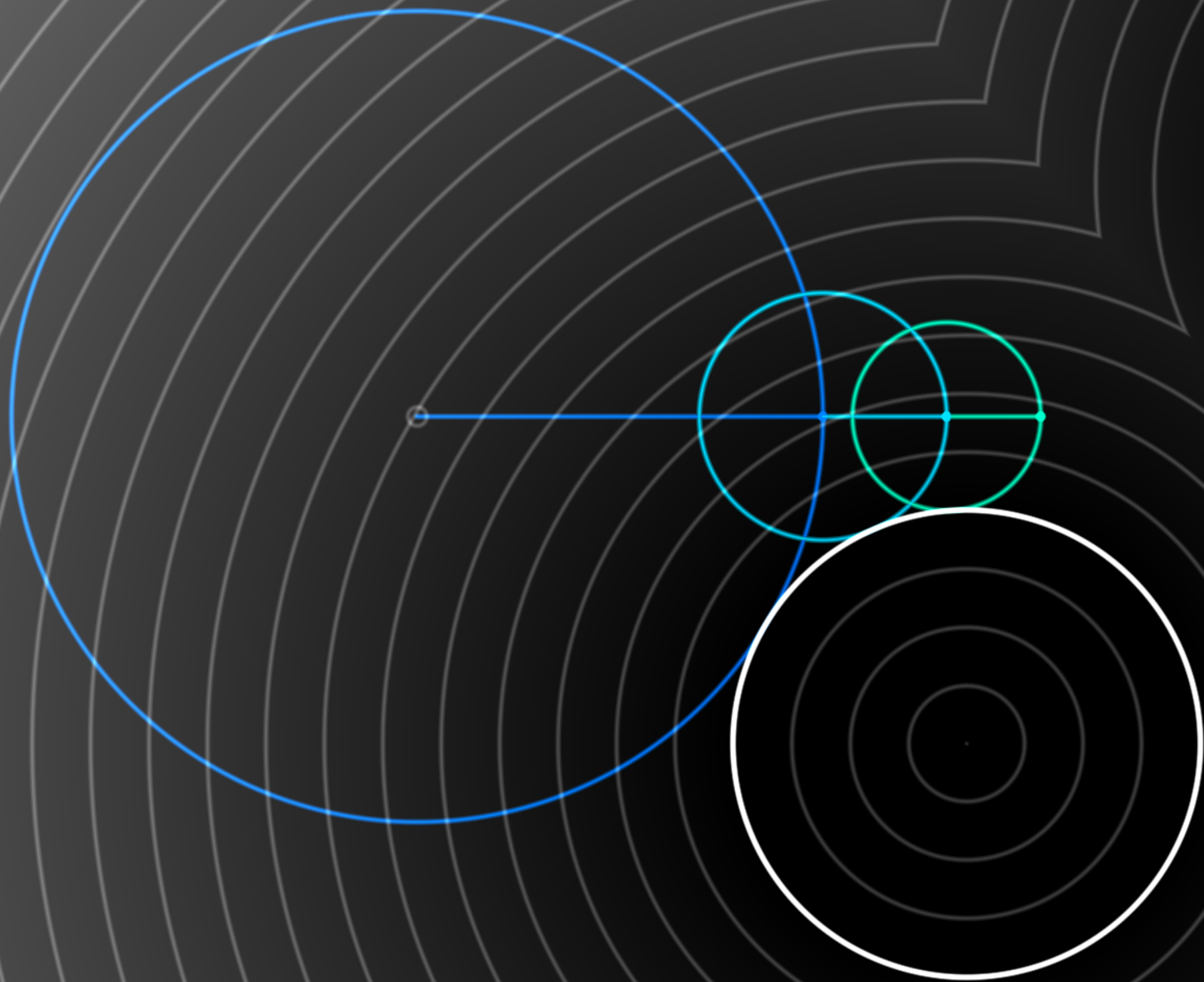
```
vec2 origin          = vec2(0.);  
vec2 position        = origin;  
  
float surface_threshold = .001;  
for(int i = 0; i < 128; i++)  
{  
    float distance_to_surface = map(position);  
    if(distance_to_surface < surface_threshold) break;  
    position += direction * distance_to_surface;  
}  
float distance_to_scene = distance(origin, position);
```

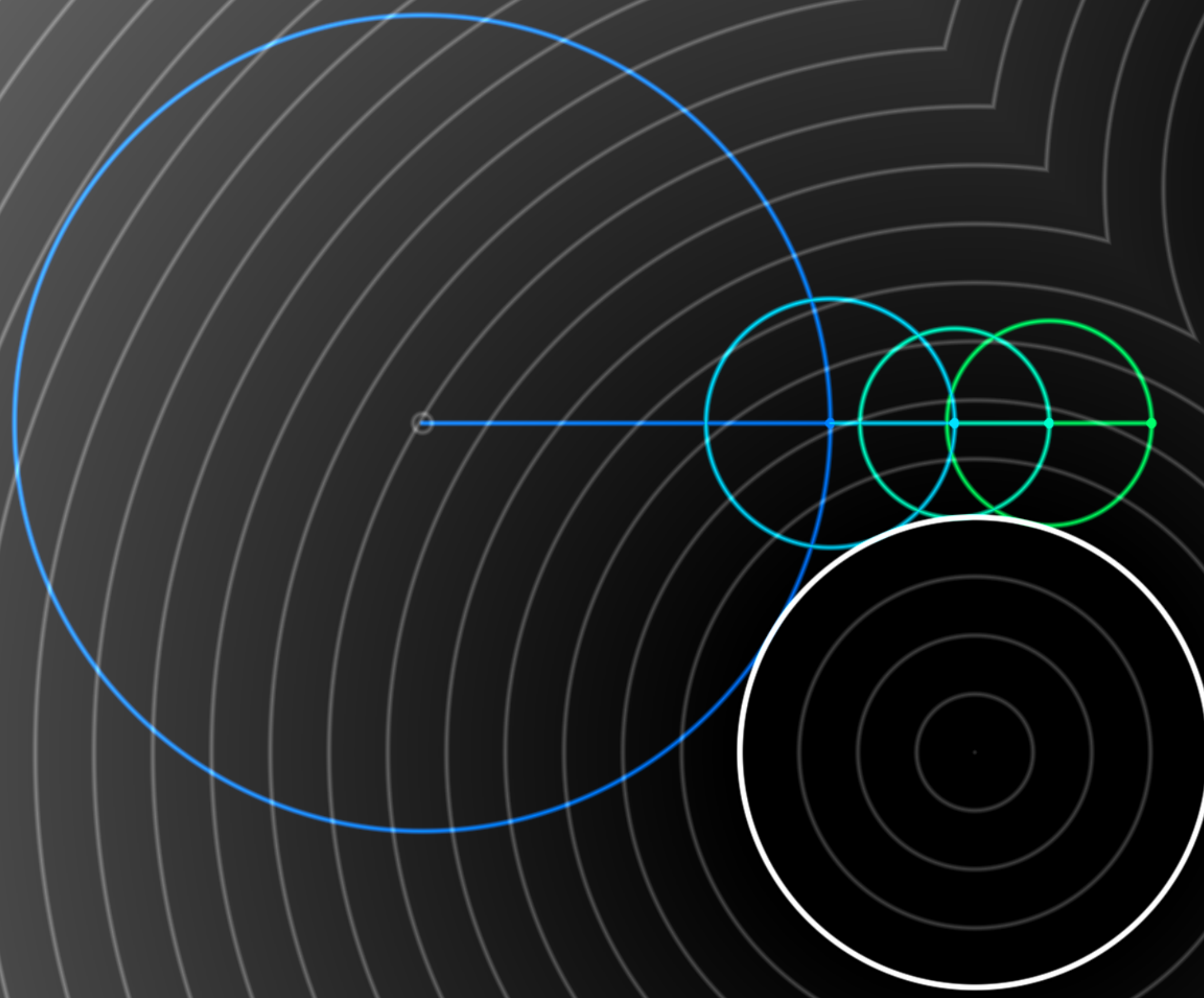



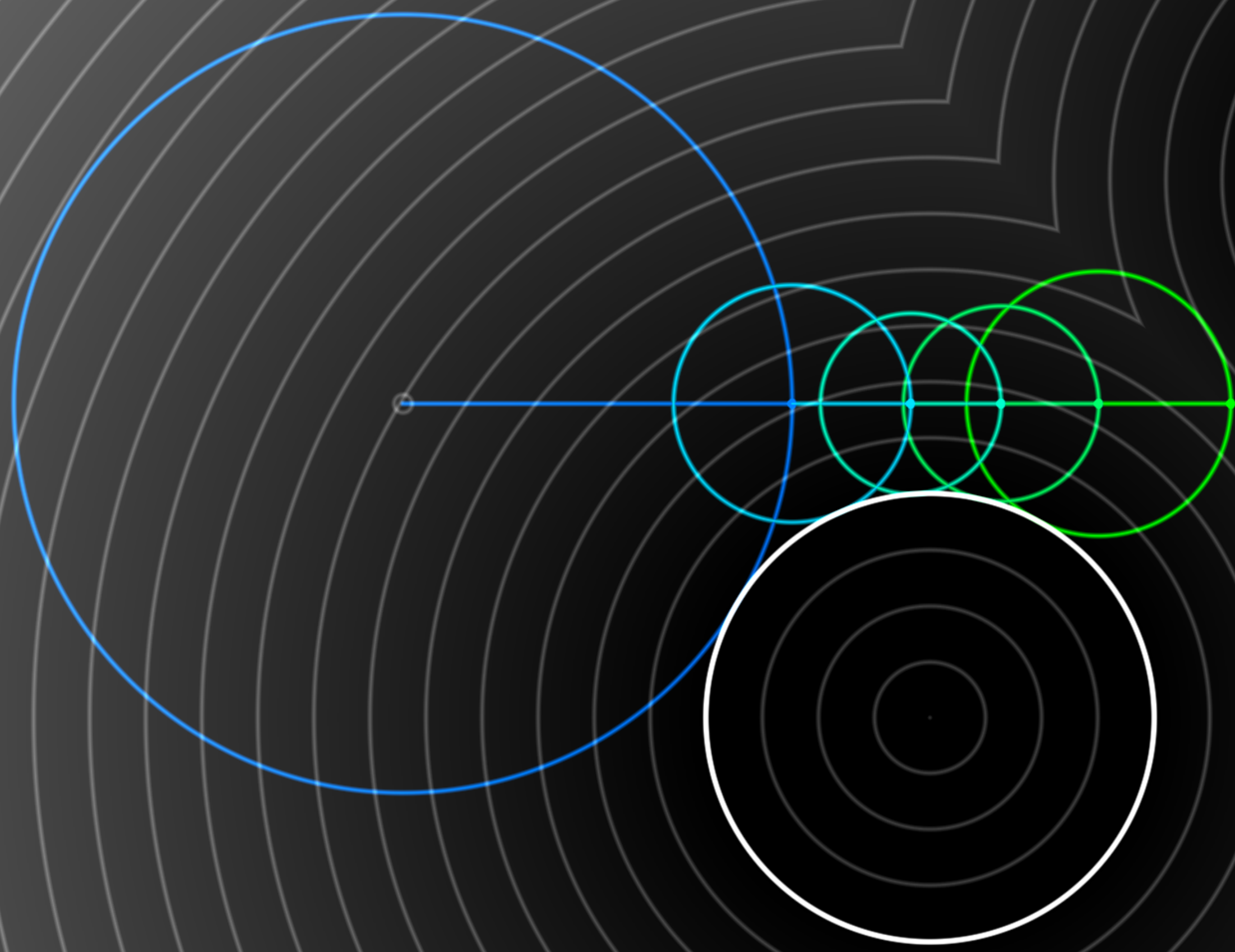

```
vec2 origin          = vec2(0.);  
vec2 position        = origin;  
  
float surface_threshold = .001;  
for(int i = 0; i < 128; i++)  
{  
    float distance_to_surface = map(position);  
    if(distance_to_surface < surface_threshold) break;  
    position += direction * distance_to_surface;  
}  
float distance_to_scene = distance(origin, position);
```

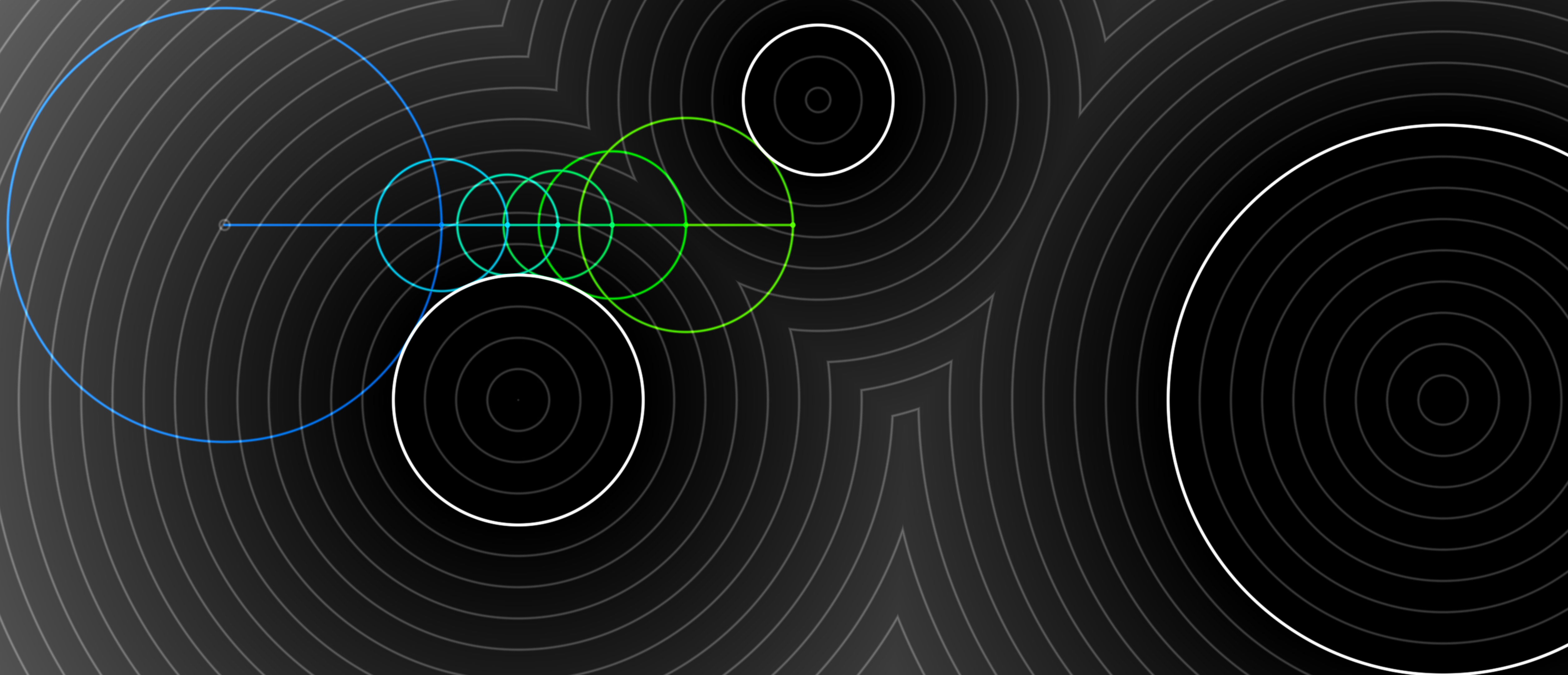



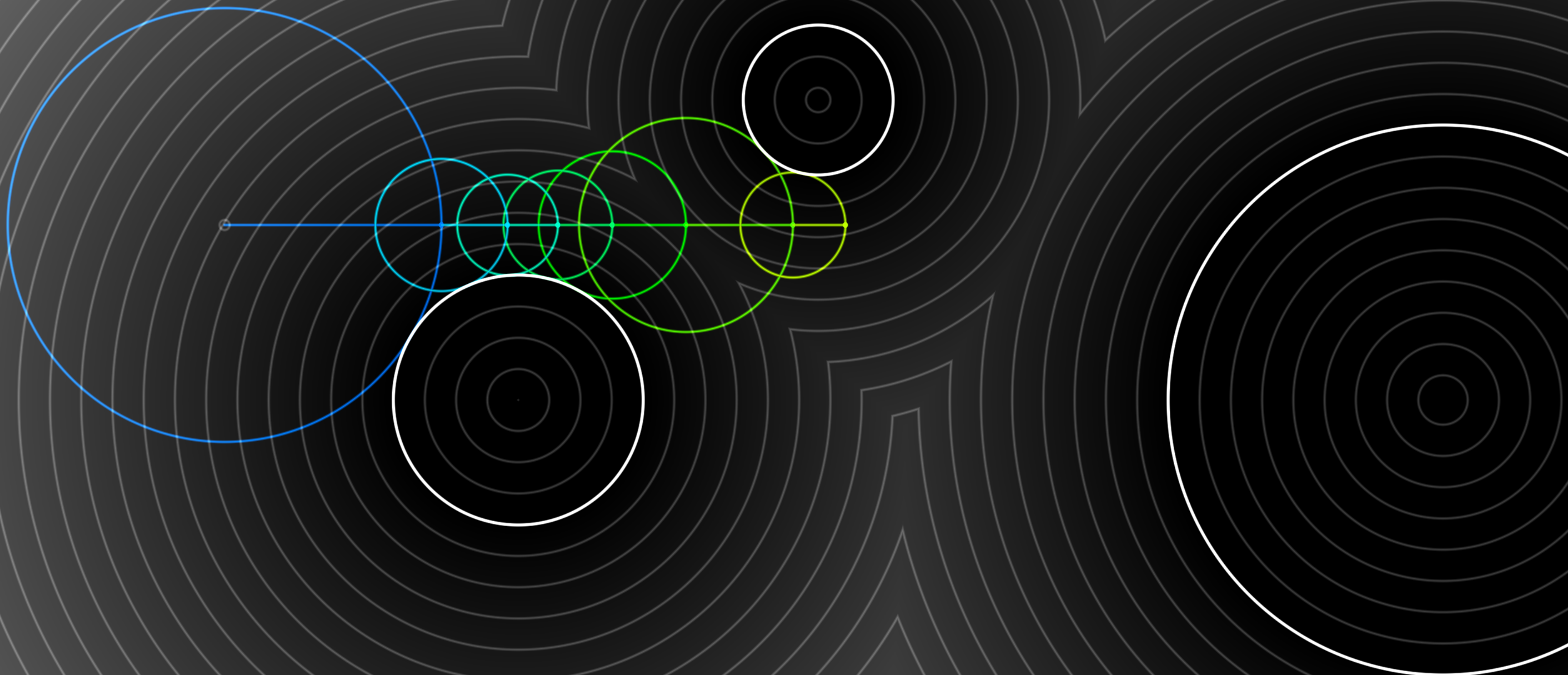

```
vec2 origin          = vec2(0.);  
vec2 position        = origin;  
  
float surface_threshold = .001;  
for(int i = 0; i < 128; i++)  
{  
    float distance_to_surface = map(position);  
    if(distance_to_surface < surface_threshold) break;  
    position += direction * distance_to_surface;  
}  
float distance_to_scene = distance(origin, position);
```

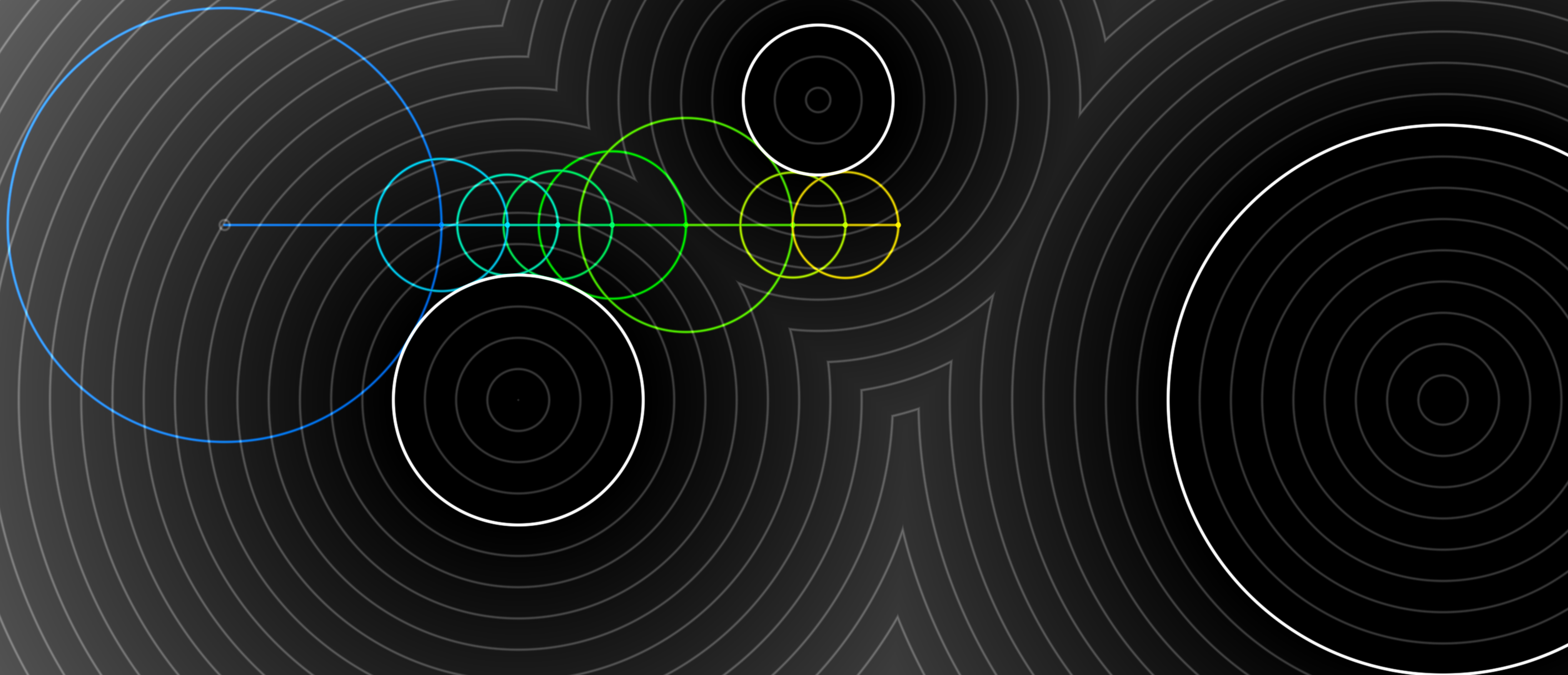



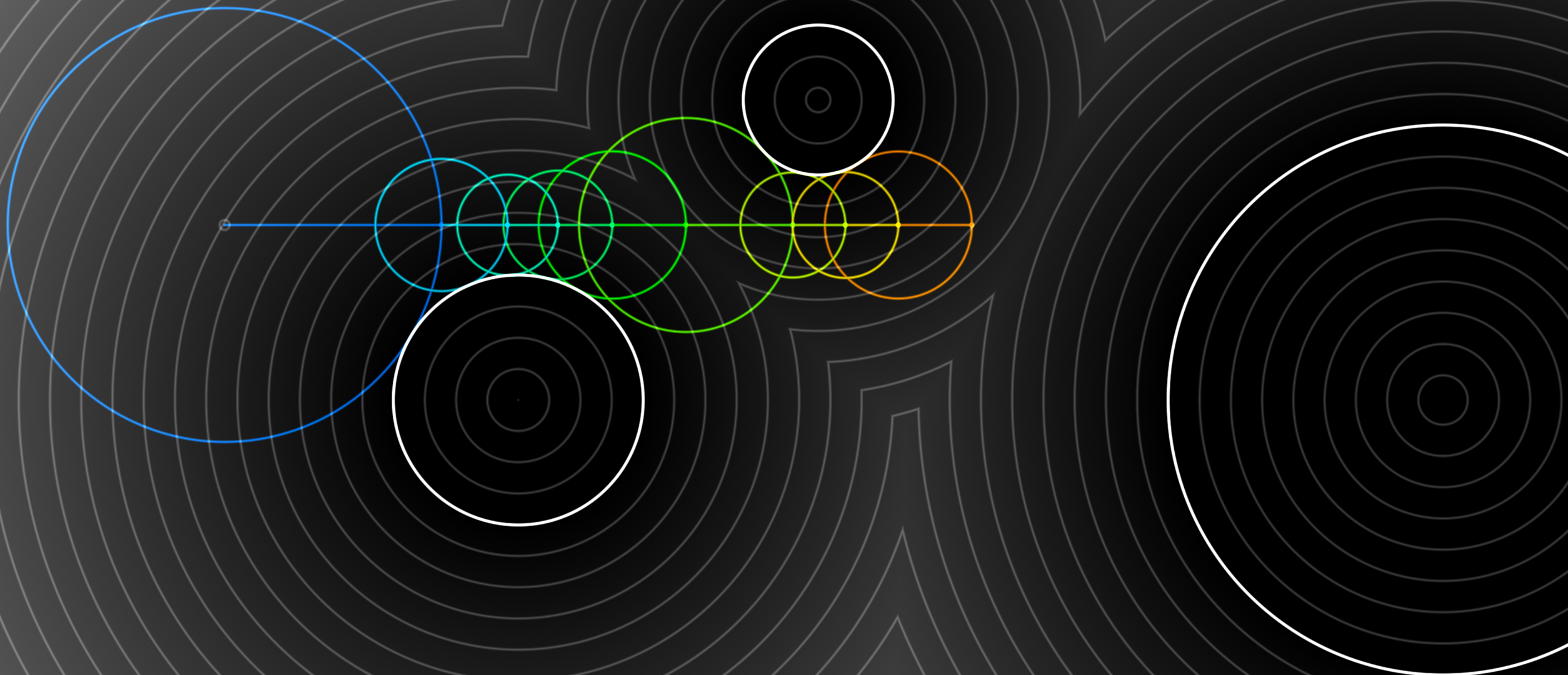


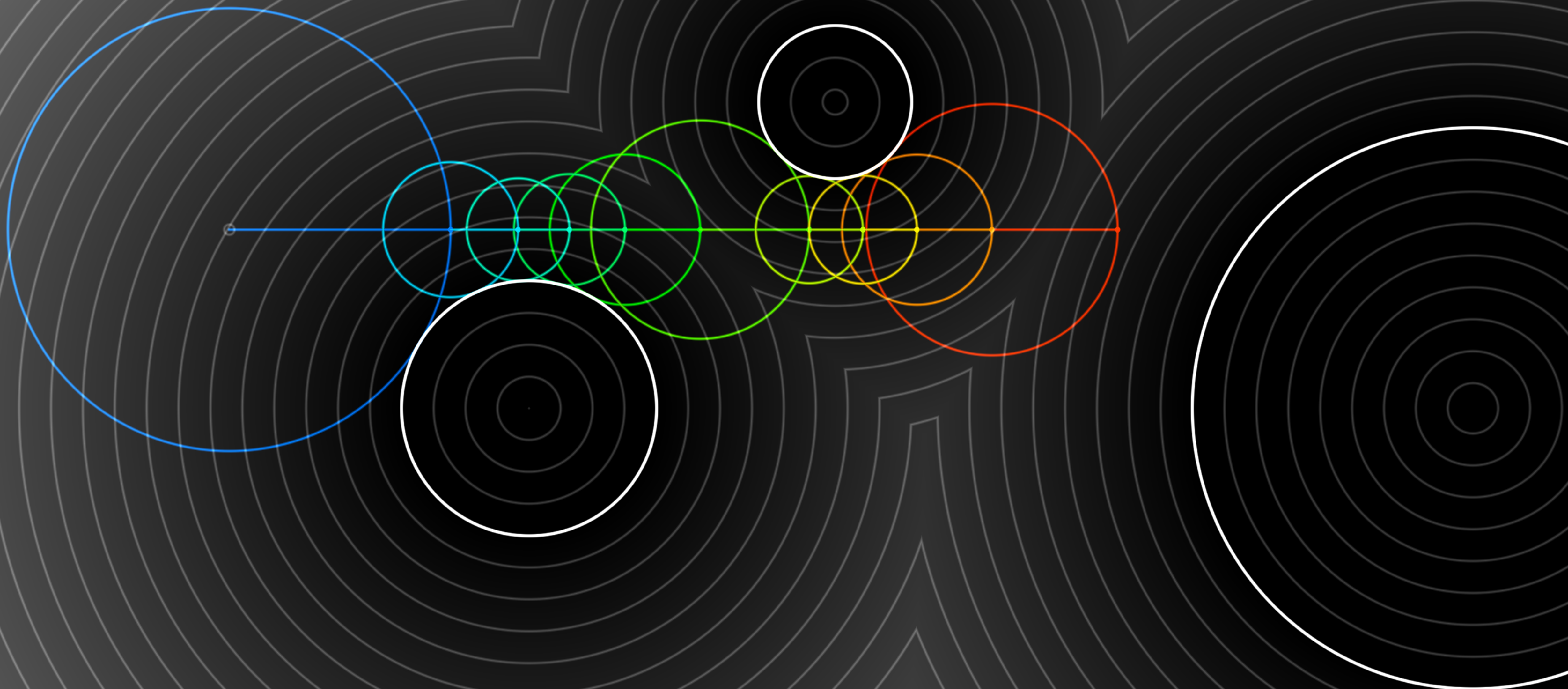


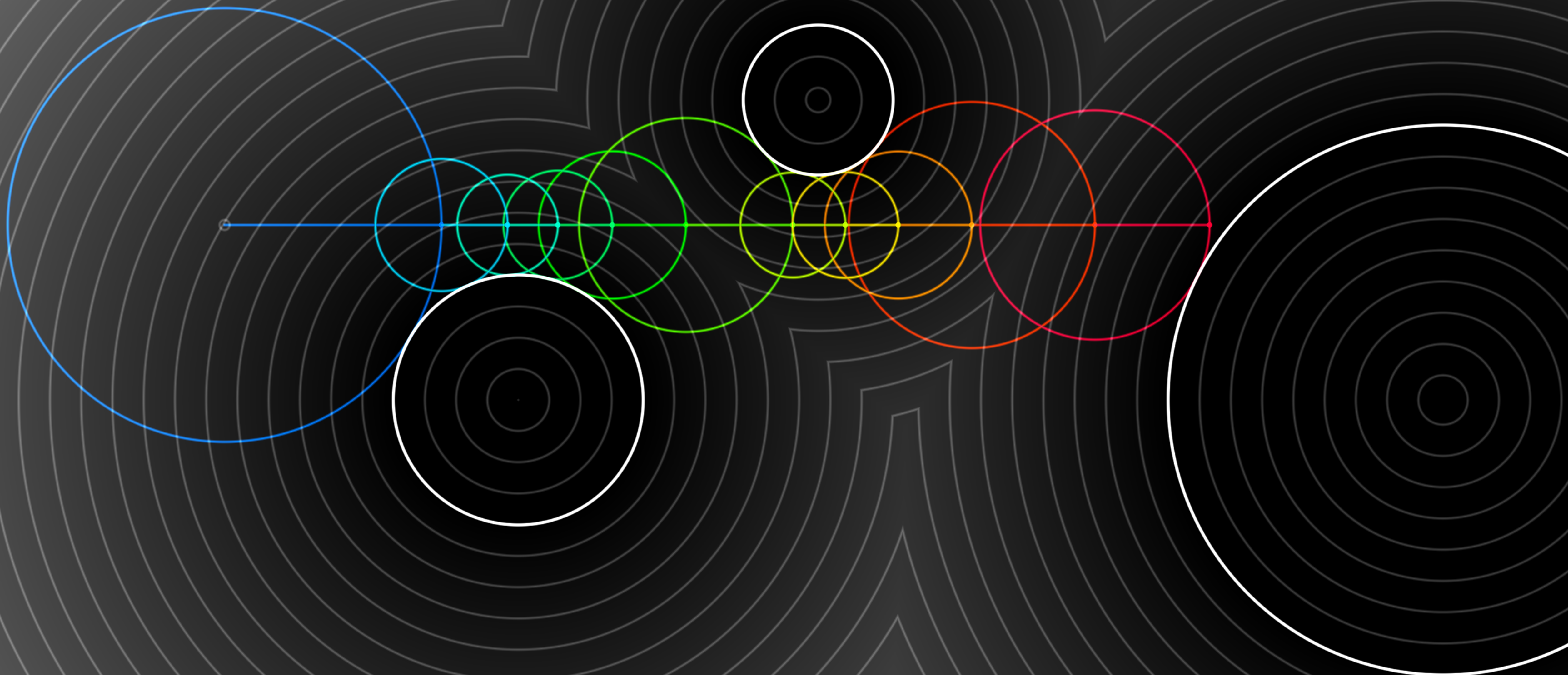


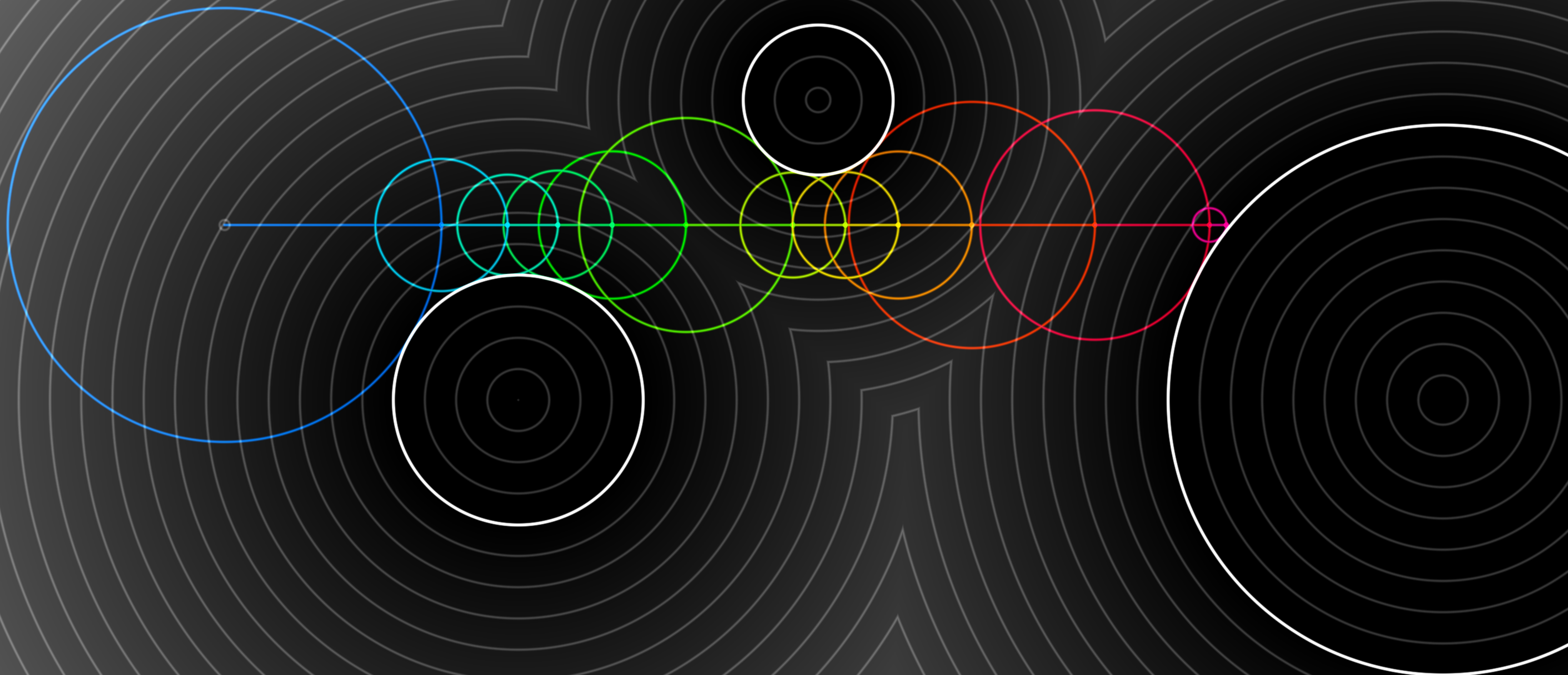




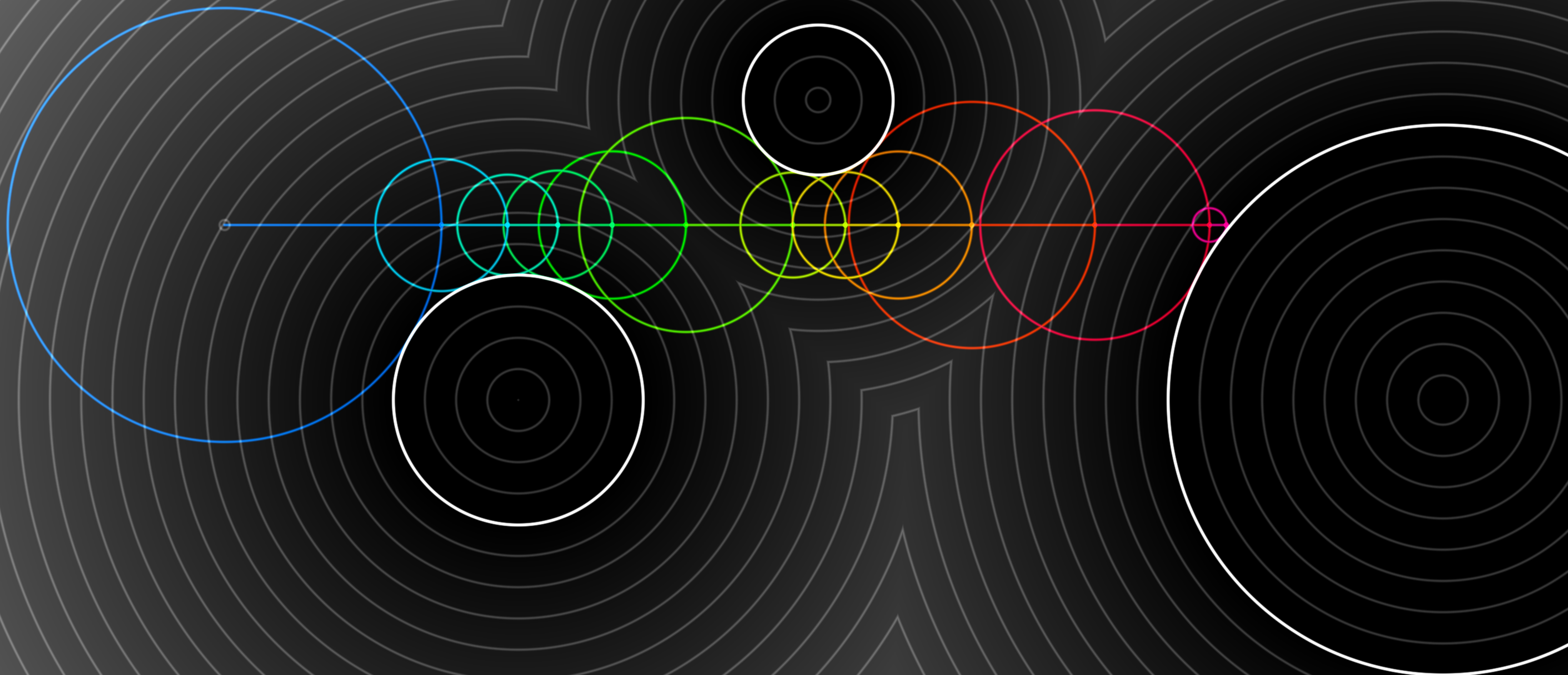


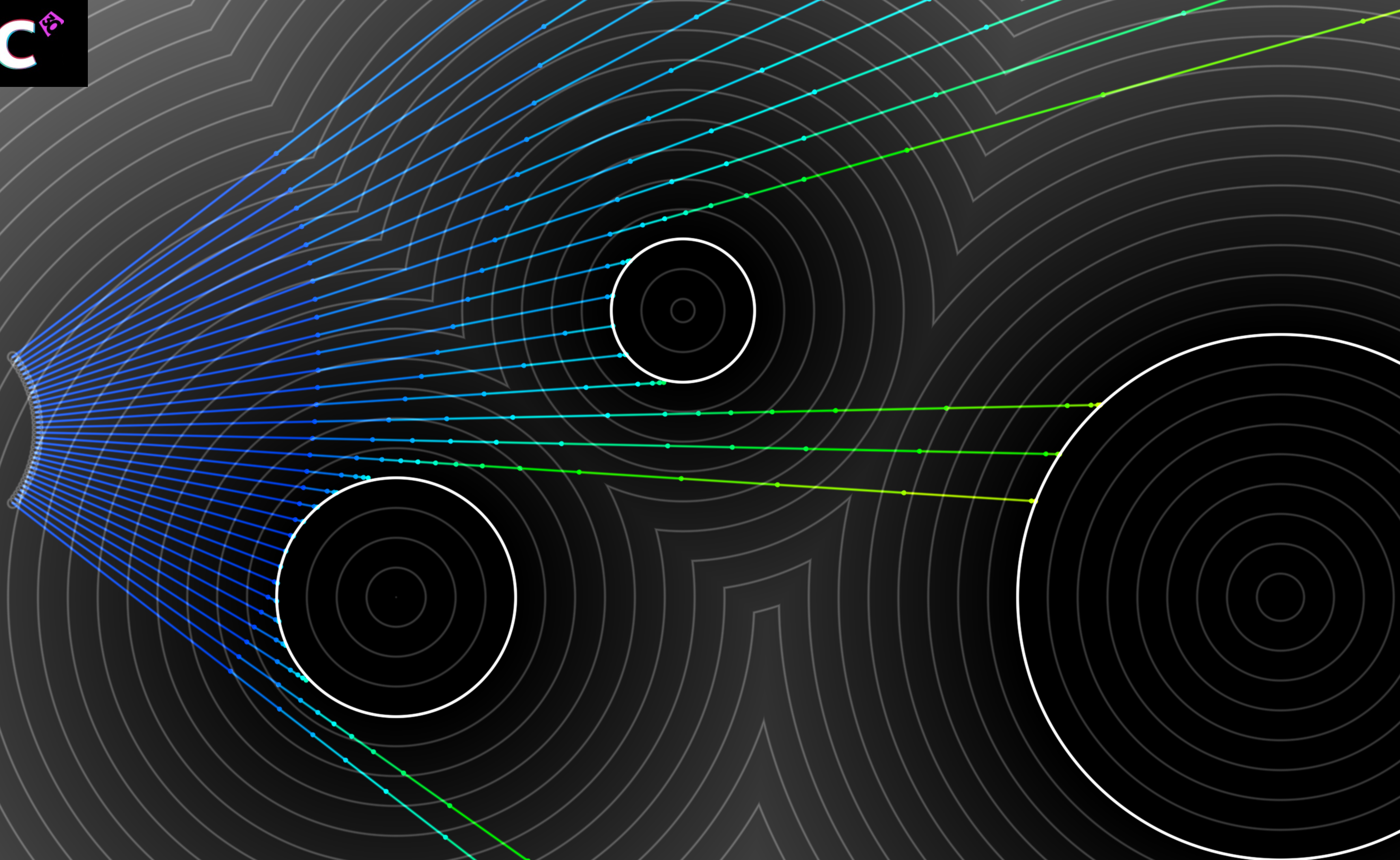


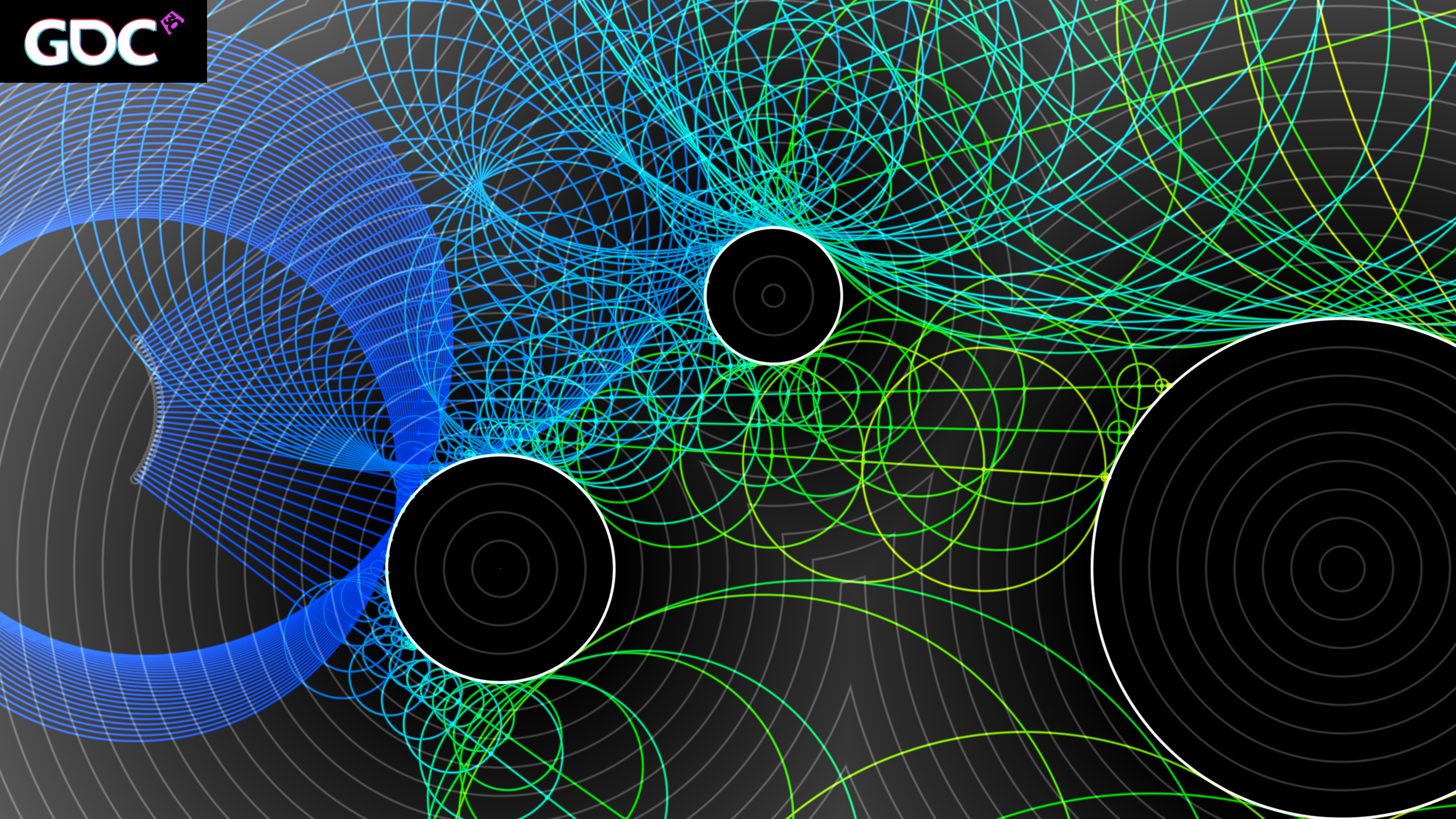




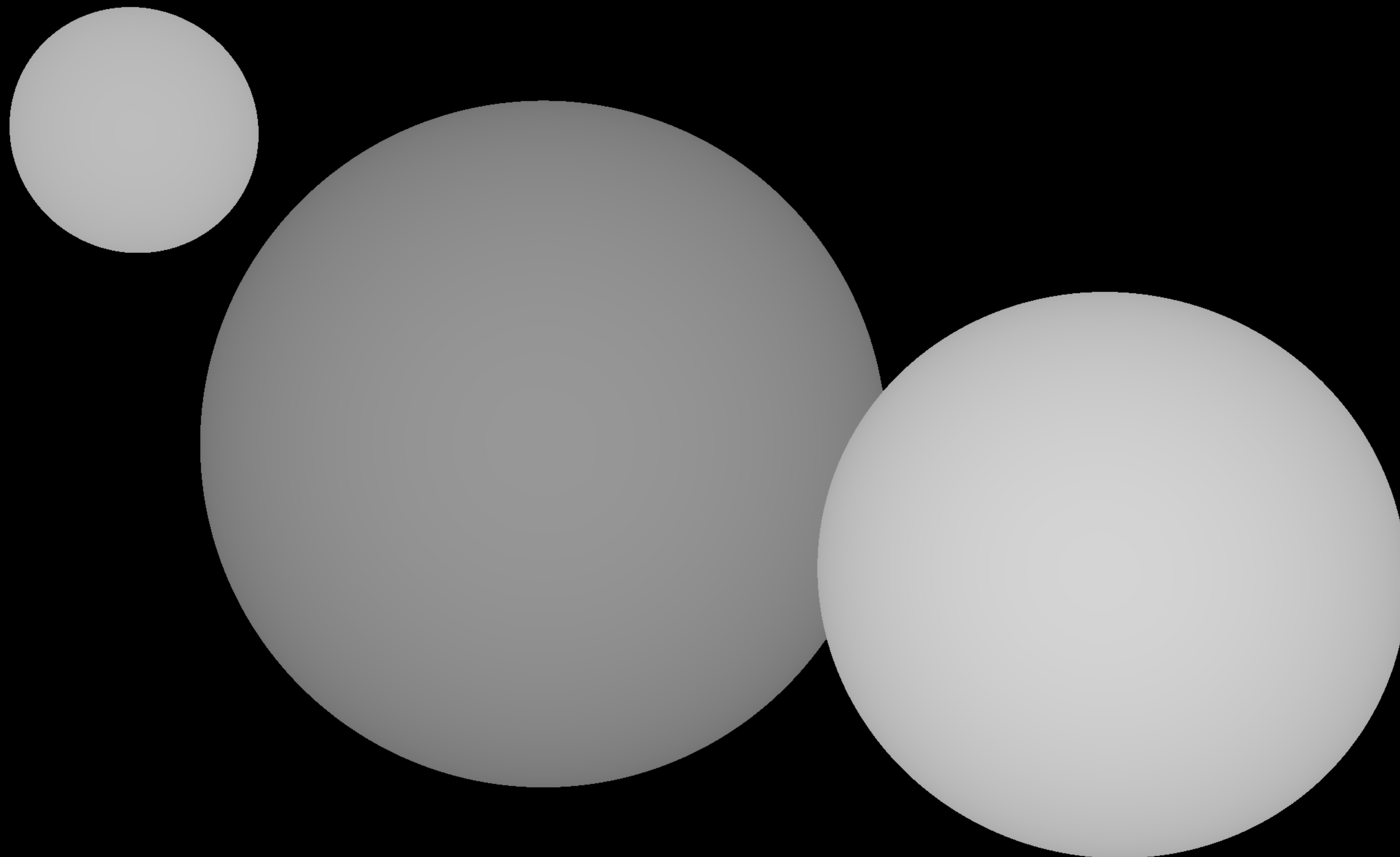

```
vec2 origin          = vec2(0.);  
vec2 position        = origin;  
  
float surface_threshold = .001;  
for(int i = 0; i < 128; i++)  
{  
    float distance_to_surface = map(position);  
    if(distance_to_surface < surface_threshold) break;  
    position += direction * distance_to_surface;  
}  
  
float distance_to_scene = distance(origin, position);
```






```
vec2 origin          = vec2(0.);  
vec2 position        = origin;  
  
float surface_threshold = .001;  
for(int i = 0; i < 128; i++)  
{  
    float distance_to_surface = map(position);  
    if(distance_to_surface < surface_threshold) break;  
    position += direction * distance_to_surface;  
}  
  
float distance_to_scene = distance(origin, position);
```



Depth

3.


Determine the surface is face.



(The “normal.”)

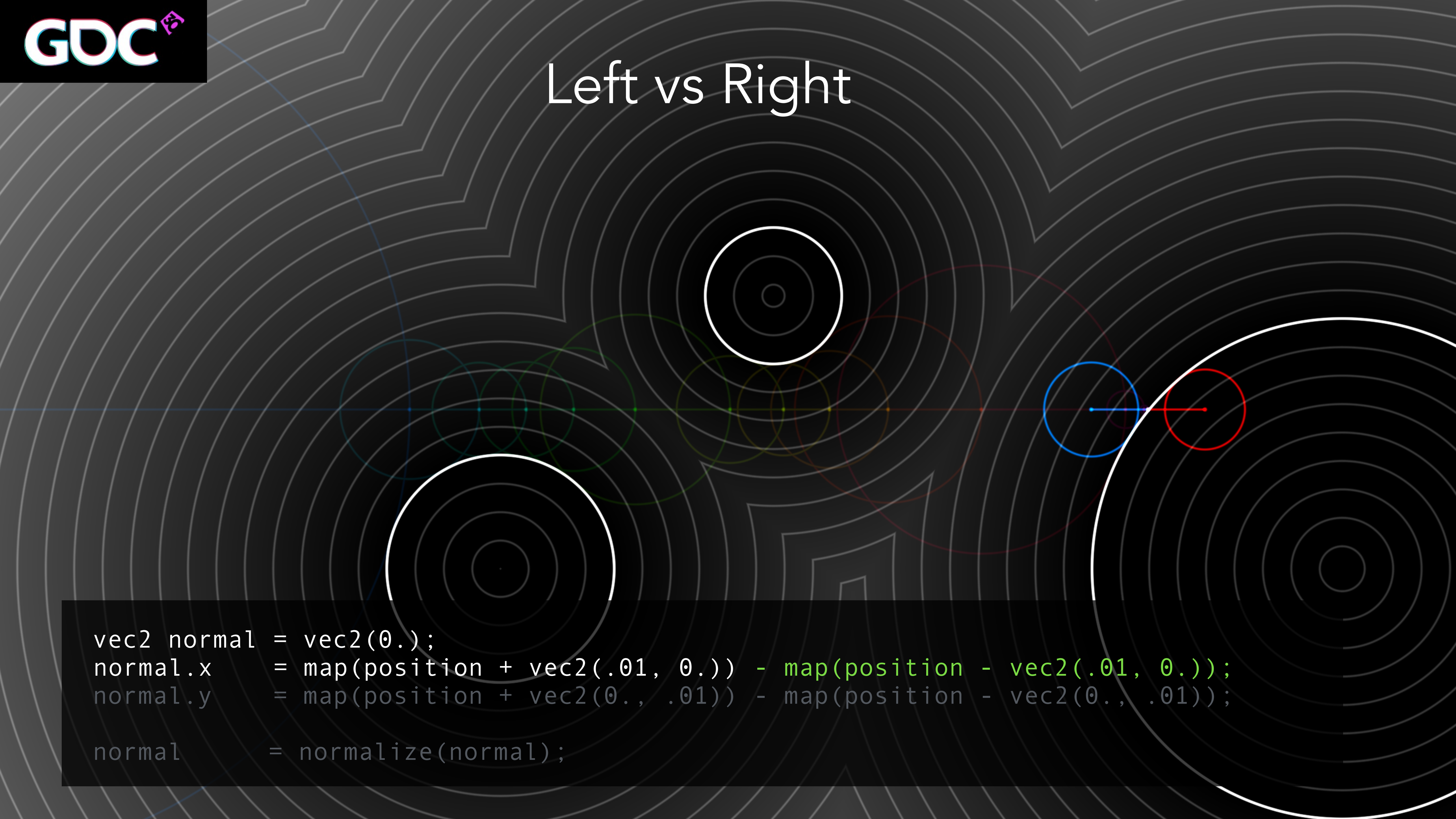
Take samples nearby the end of the ray, and compare them for the partial derivative.

Left




```
vec2 normal = vec2(0.);  
normal.x   = map(position + vec2(.01, 0.)) - map(position - vec2(.01, 0.));  
normal.y   = map(position + vec2(0., .01)) - map(position - vec2(0., .01));  
  
normal     = normalize(normal);
```


Left vs Right



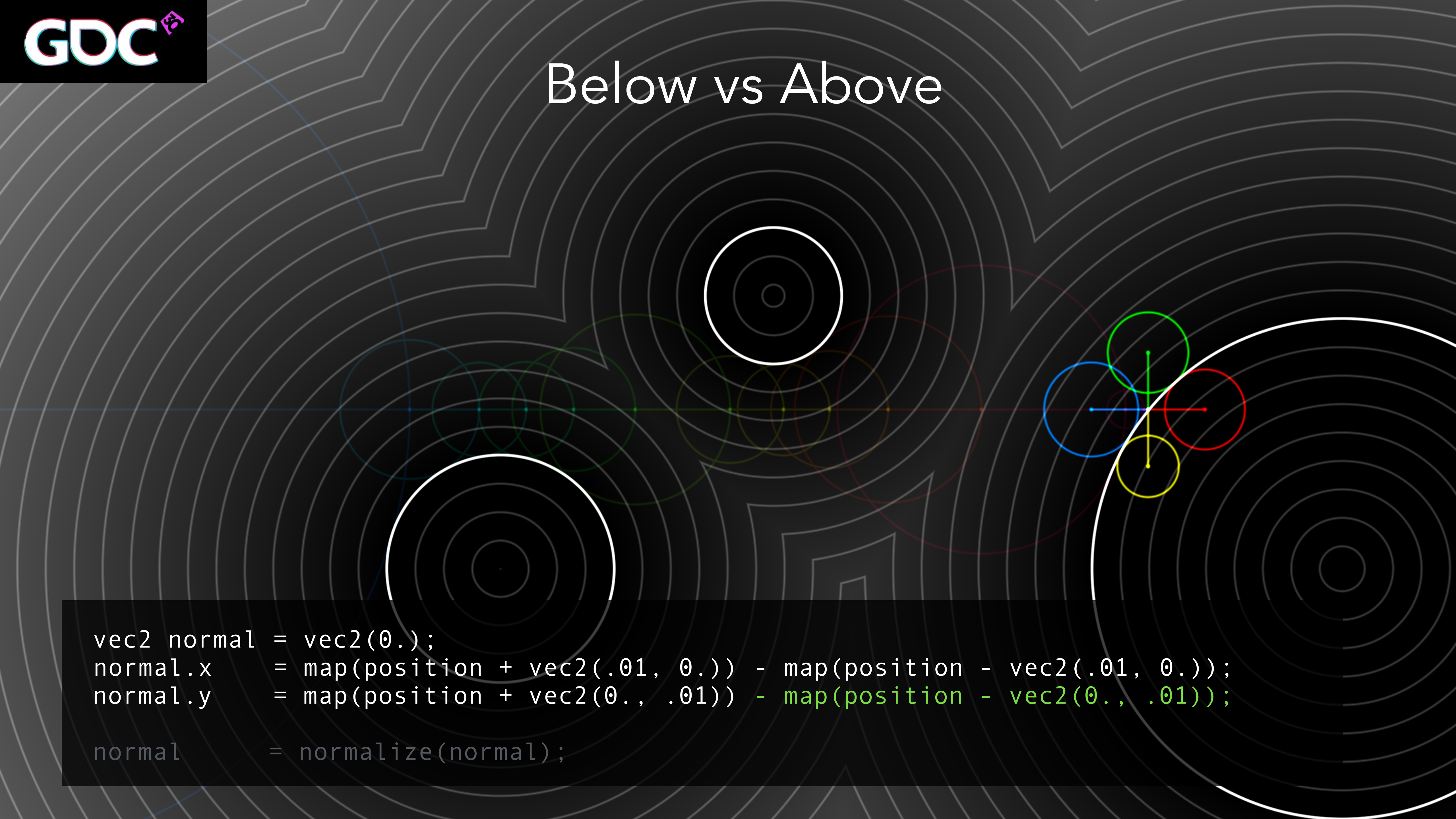
```
vec2 normal = vec2(0.);  
normal.x   = map(position + vec2(.01, 0.)) - map(position - vec2(.01, 0.));  
normal.y   = map(position + vec2(0., .01)) - map(position - vec2(0., .01));  
  
normal     = normalize(normal);
```


Below



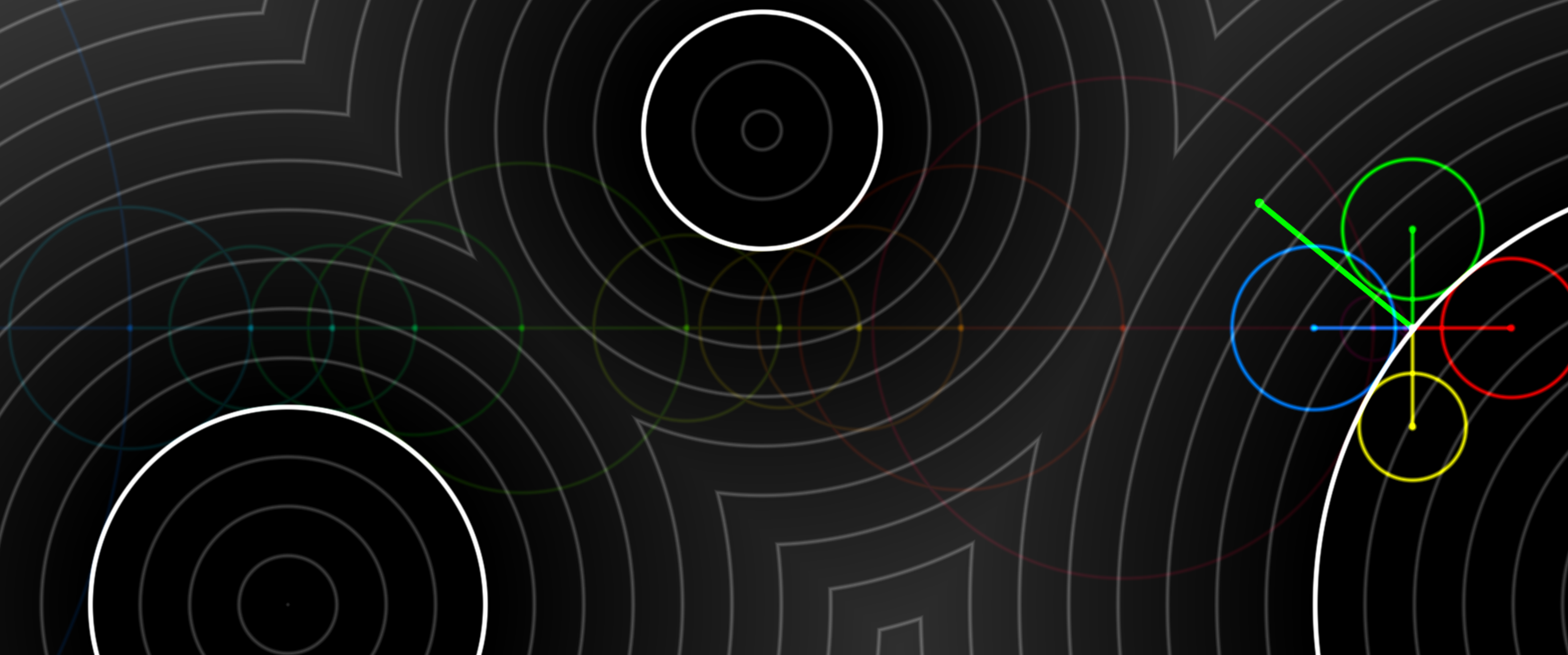
```
vec2 normal = vec2(0.);  
normal.x   = map(position + vec2(.01, 0.)) - map(position - vec2(.01, 0.));  
normal.y   = map(position + vec2(0., .01)) - map(position - vec2(0., .01));  
  
normal     = normalize(normal);
```


Below vs Above



```
vec2 normal = vec2(0.);  
normal.x   = map(position + vec2(.01, 0.)) - map(position - vec2(.01, 0.));  
normal.y   = map(position + vec2(0., .01)) - map(position - vec2(0., .01));  
  
normal     = normalize(normal);
```

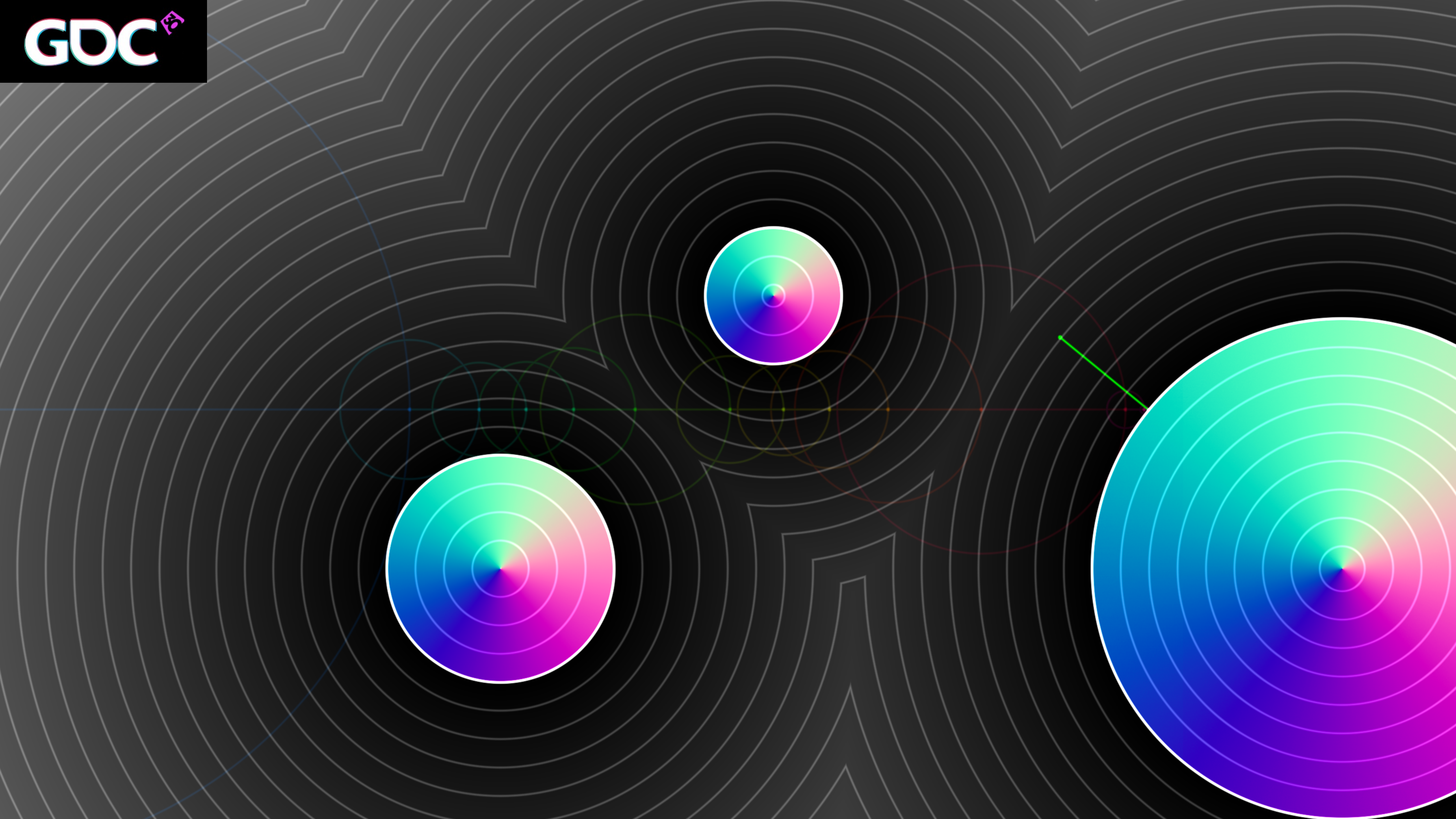

And normalize by dividing by the result of the pythagorean theorem.

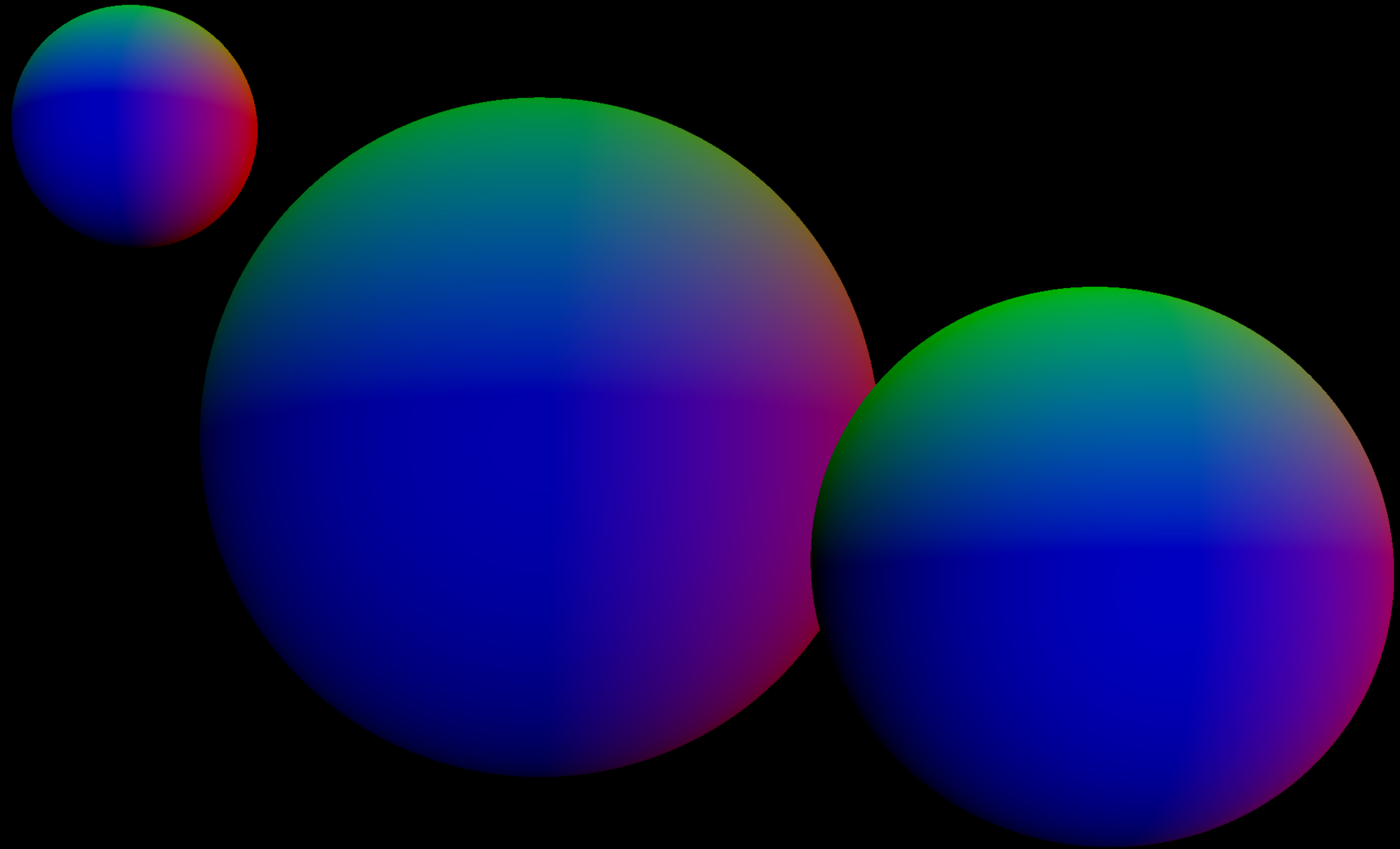


```
vec2 normal = vec2(0.);  
normal.x    = map(position + vec2(.01, 0.)) - map(position - vec2(.01, 0.));  
normal.y    = map(position + vec2(0., .01)) - map(position - vec2(0., .01));  
  
normal      = normalize(normal);
```

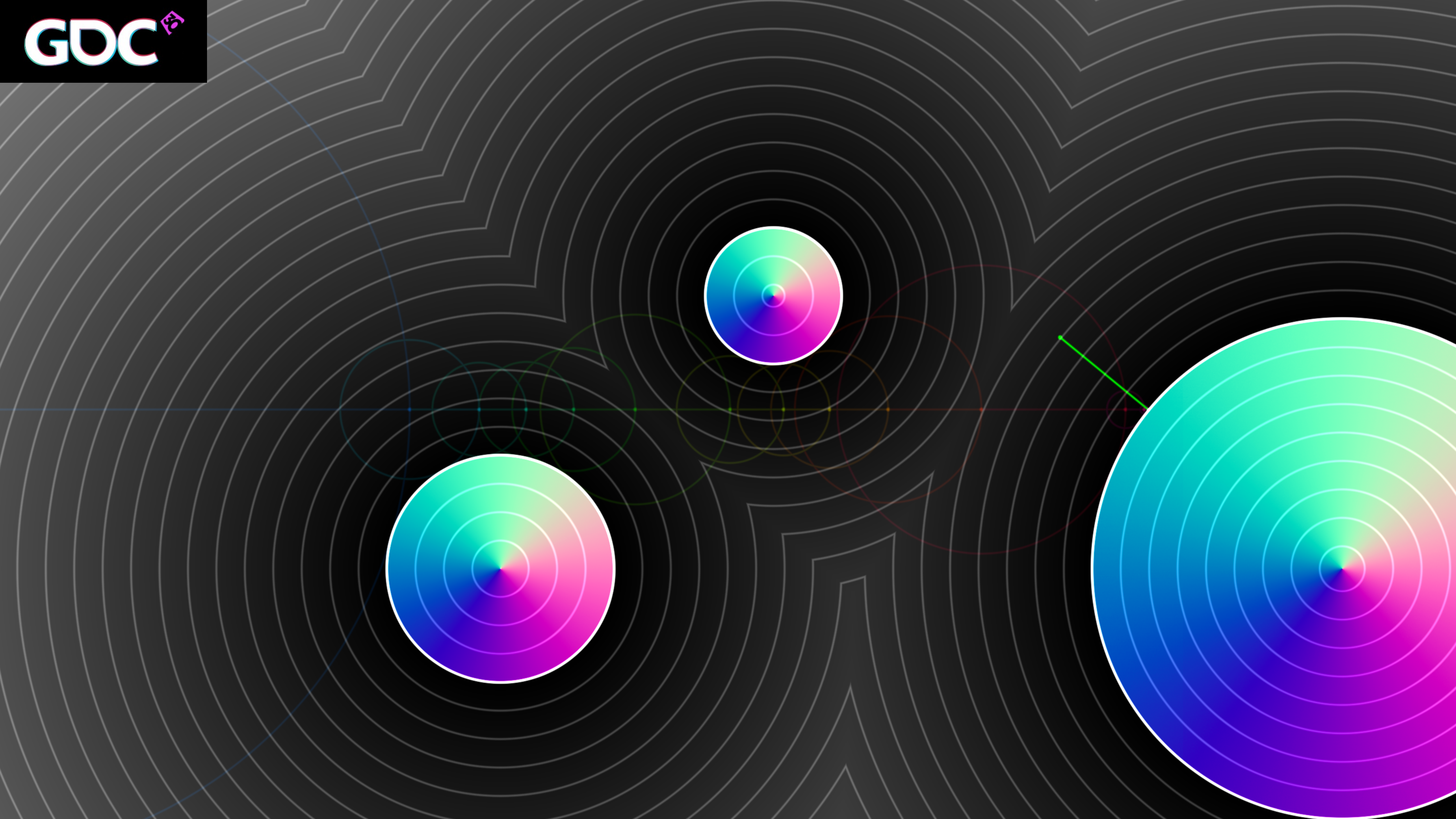

And normalize by dividing by the length.

```
vec2 normal = vec2(0.);  
normal.x    = map(position + vec2(.01, 0.)) - map(position - vec2(.01, 0.));  
normal.y    = map(position + vec2(0., .01)) - map(position - vec2(0., .01));  
  
normal      = normalize(normal);
```

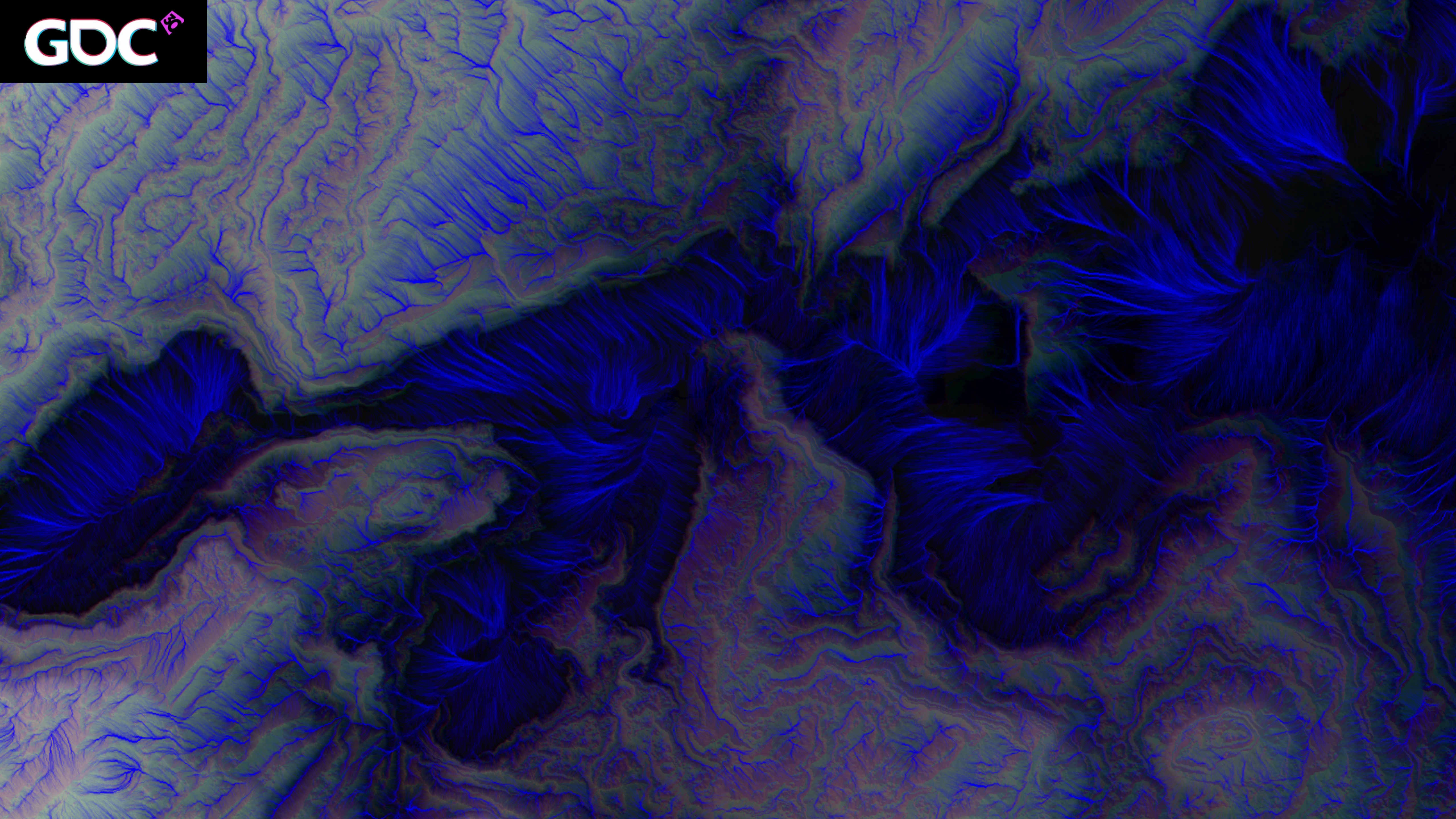


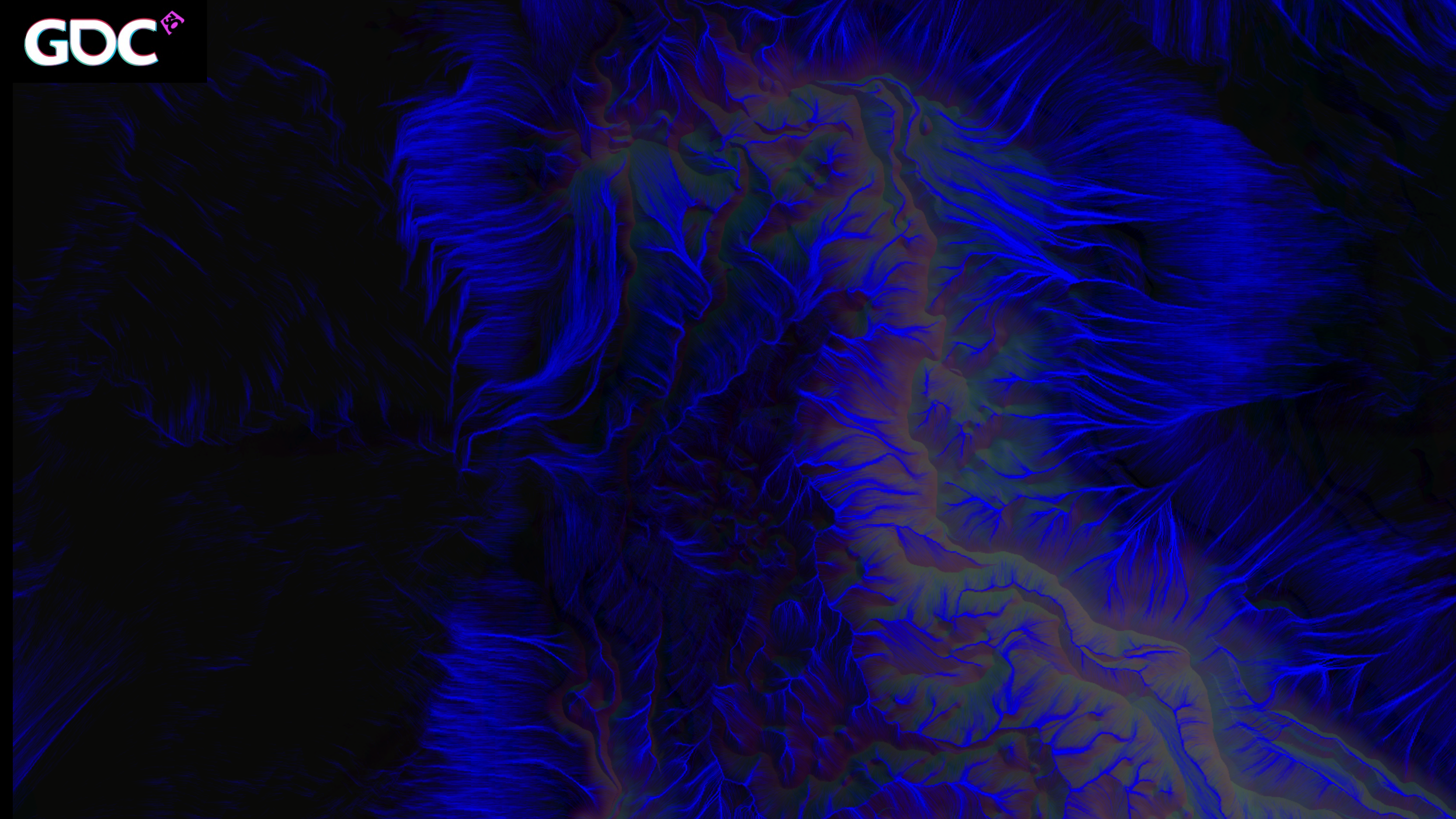
Normals

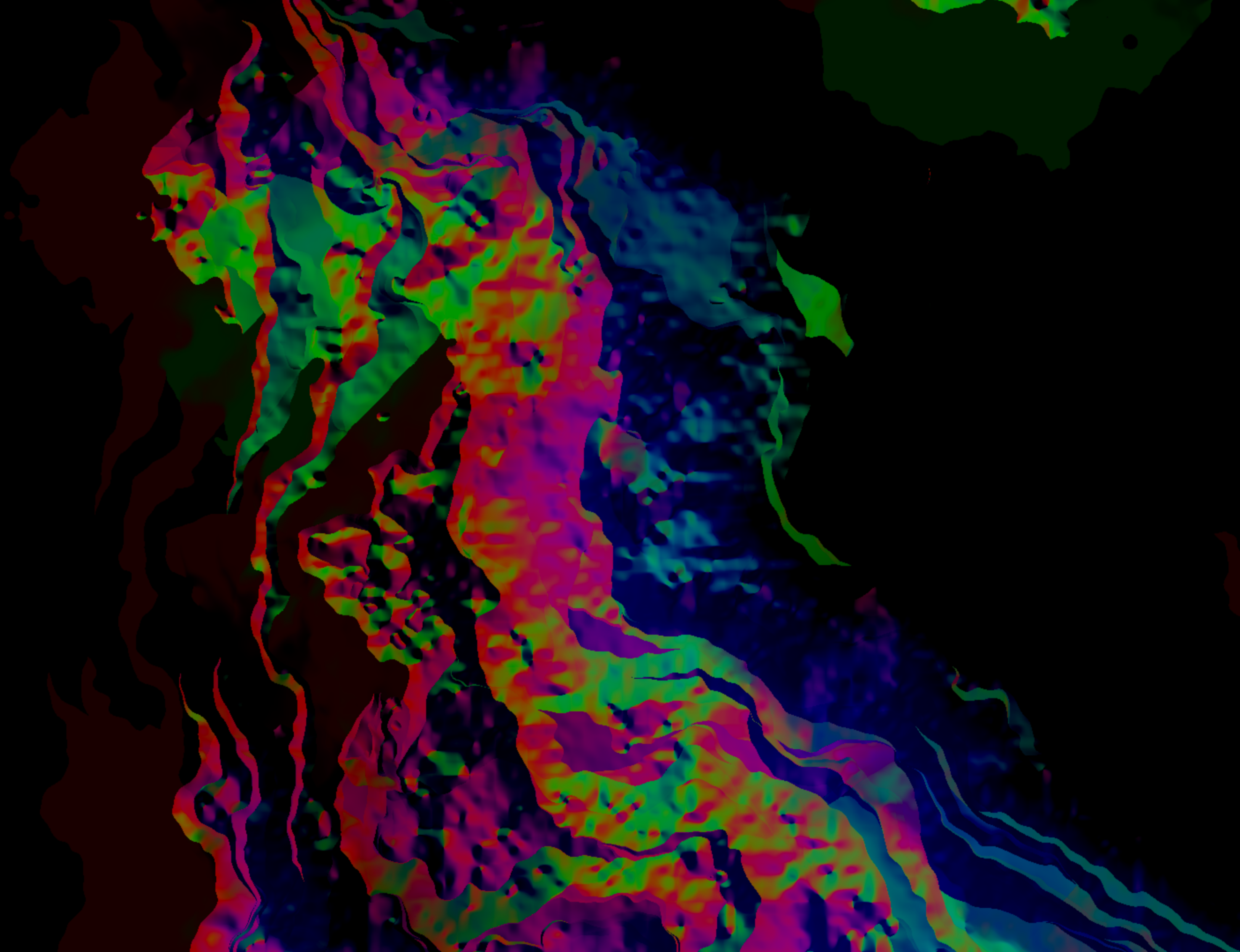


Normals can be sampled from anywhere in the field,
not just the surfaces.

River Simulation



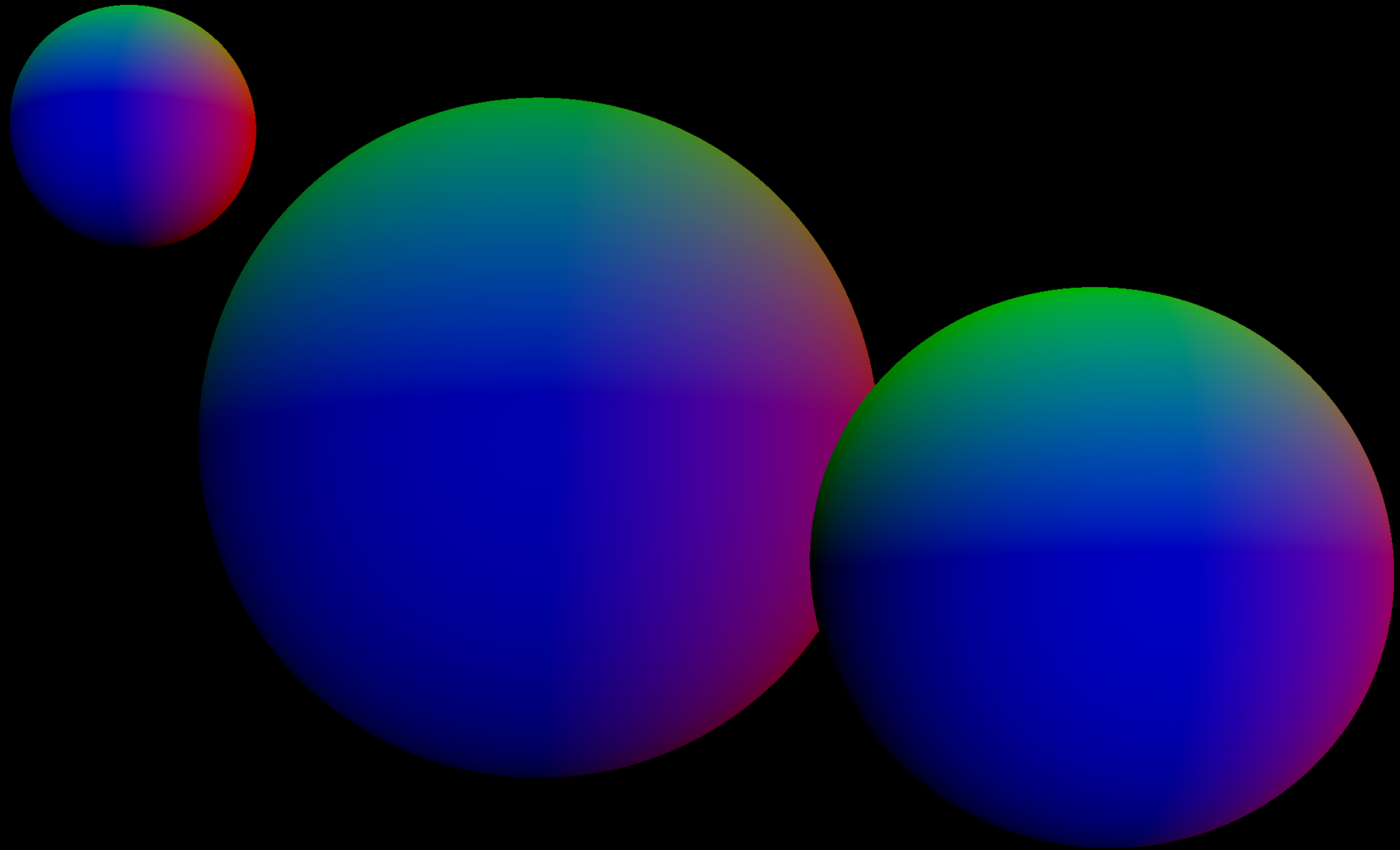




4.




Add light.



Normals

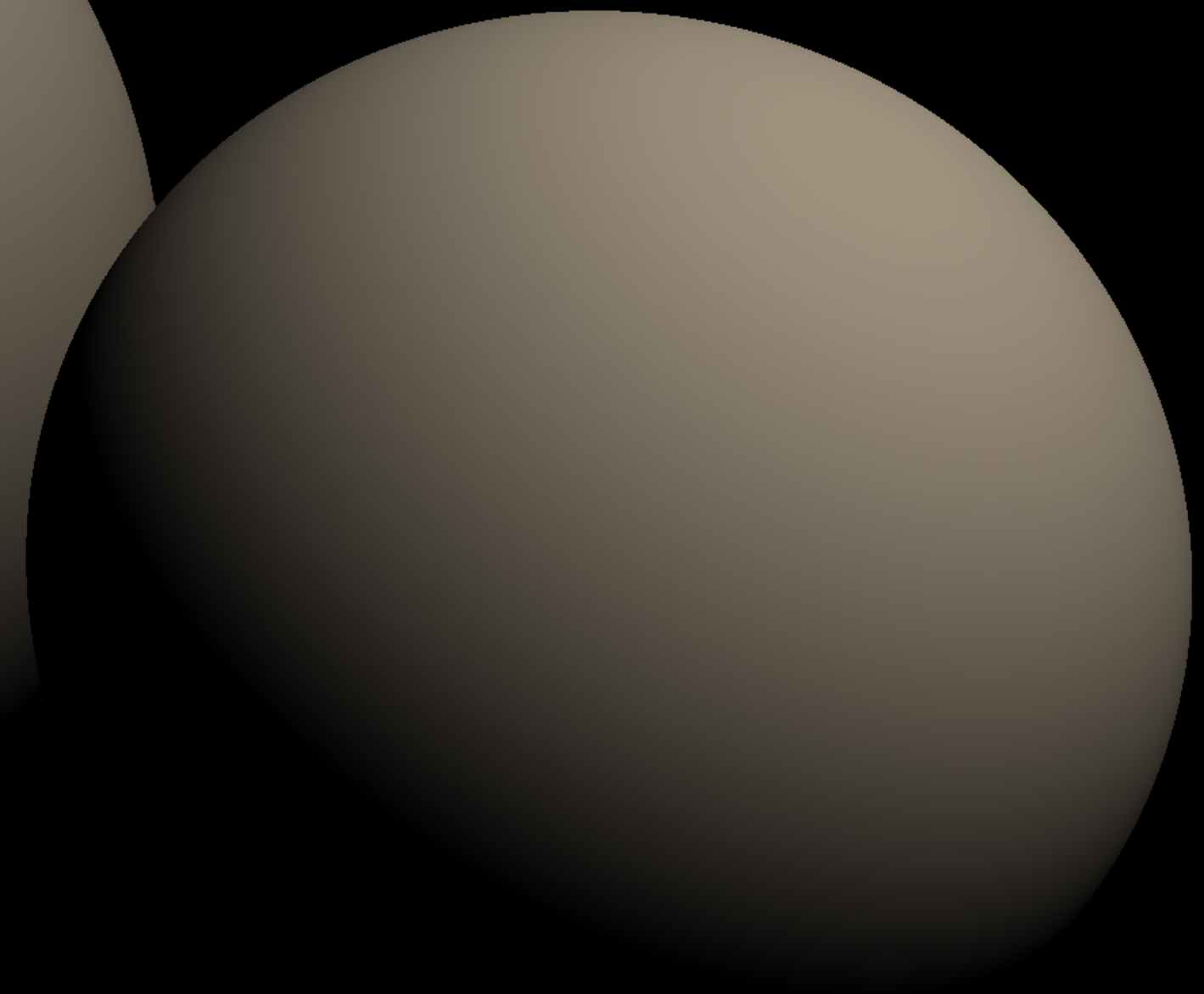
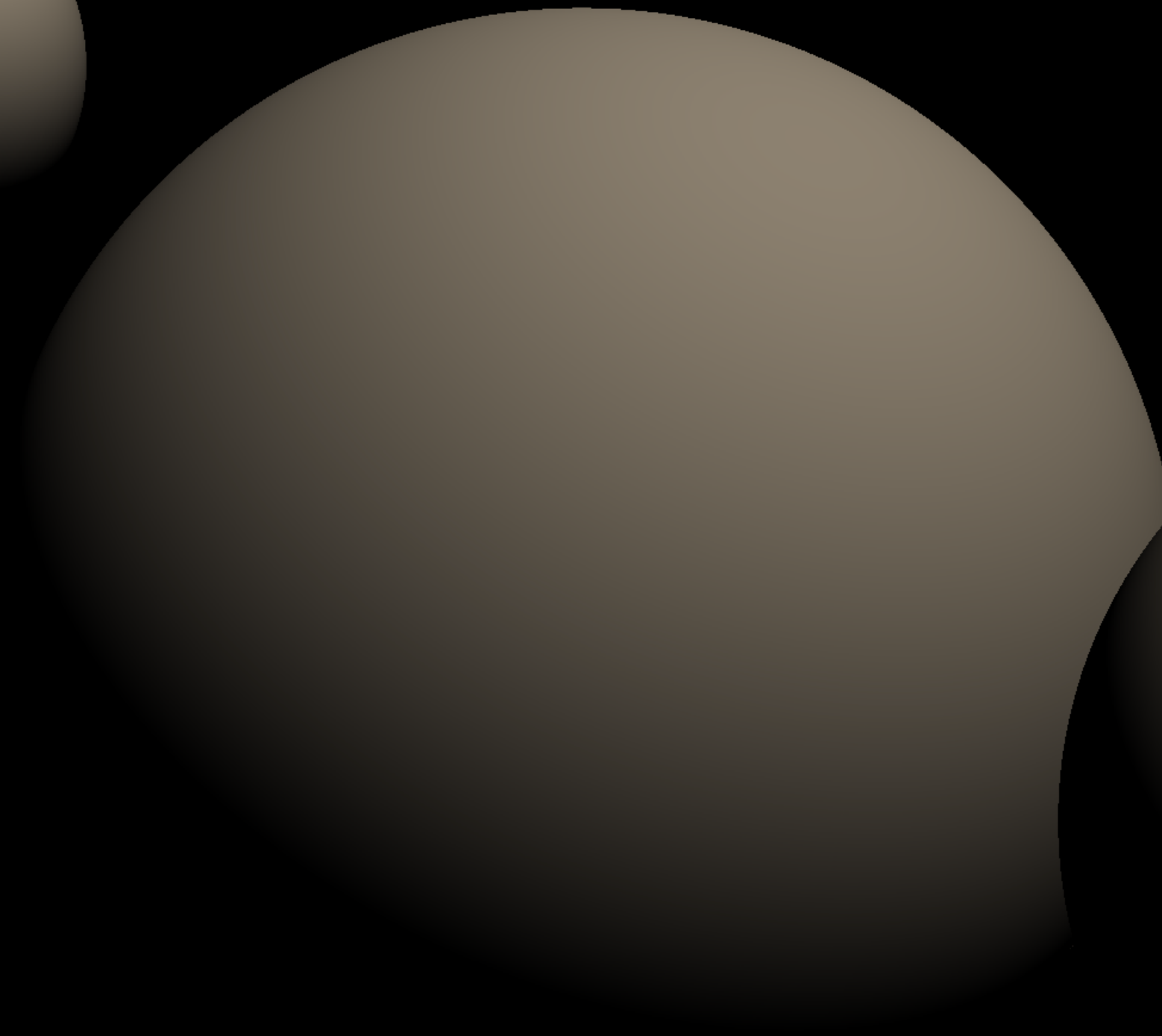

```
vec2 light_position      = vec2(.6, 1.);  
vec3 light_color         = vec3(.87, .8, .7);  
vec2 light_direction    = normalize(light_position-position);  
  
float diffuse_light = dot(normal, light_direction);
```

```
vec2 light_position      = vec2(.6, 1.);  
vec3 light_color        = vec3(.87, .8, .7);  
vec2 light_direction    = normalize(light_position-position);  
  
float diffuse_light = dot(normal, light_direction);
```



```
vec2 light_position      = vec2(.6, 1.);  
vec3 light_color         = vec3(.87, .8, .7);  
vec2 light_direction    = normalize(light_position-position);  
  
float diffuse_light = dot(normal, light_direction);
```

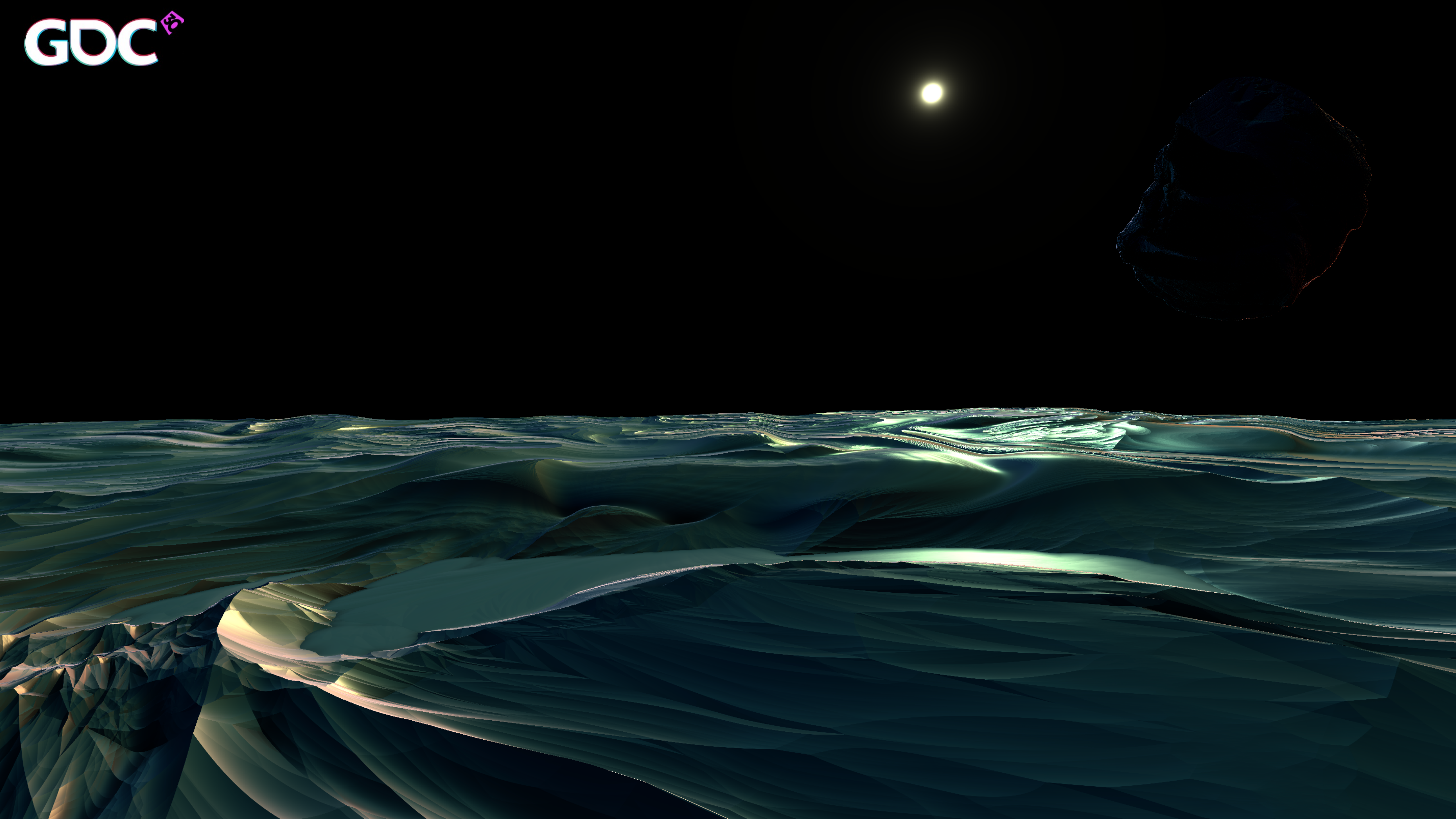



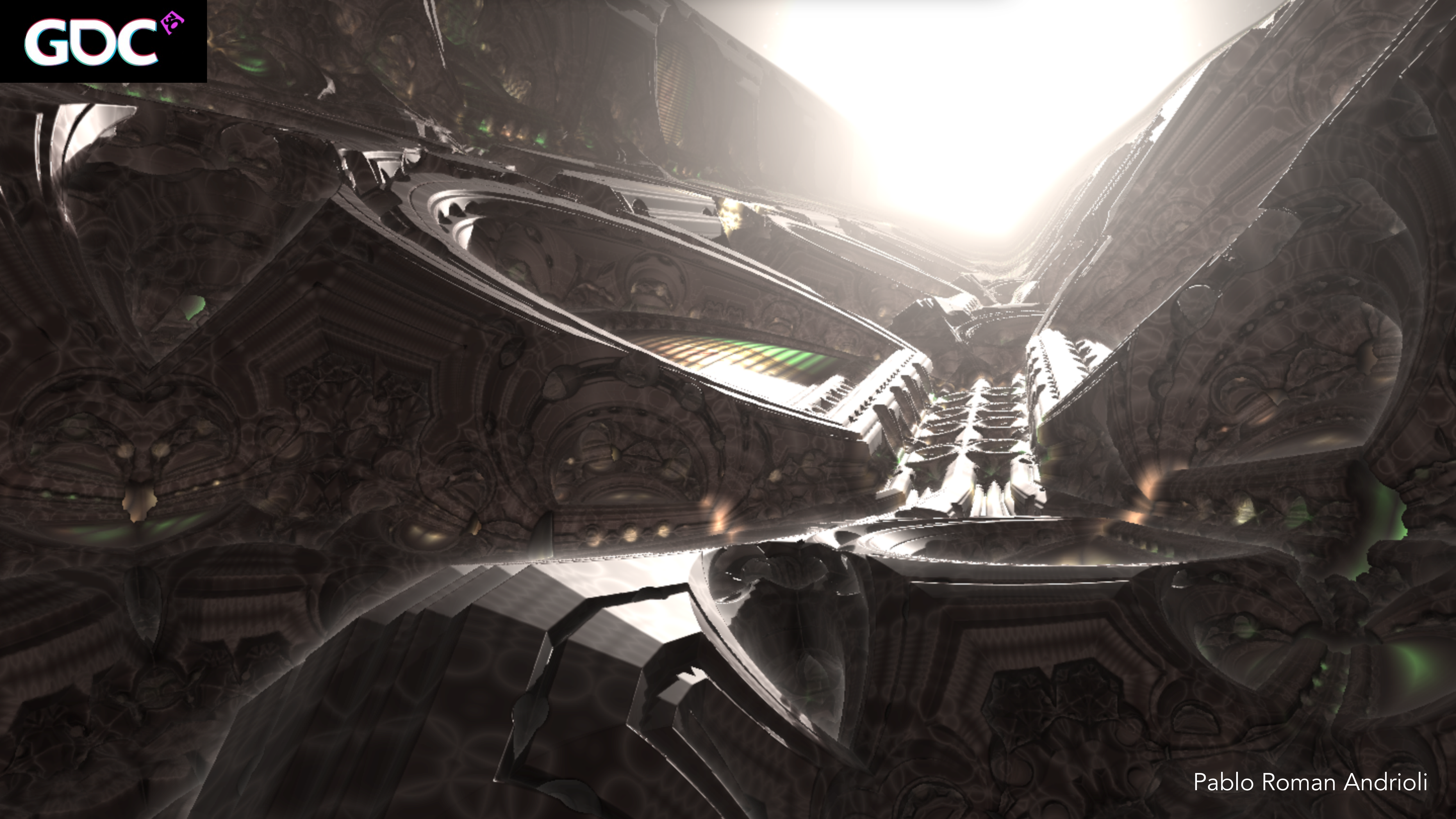
Light

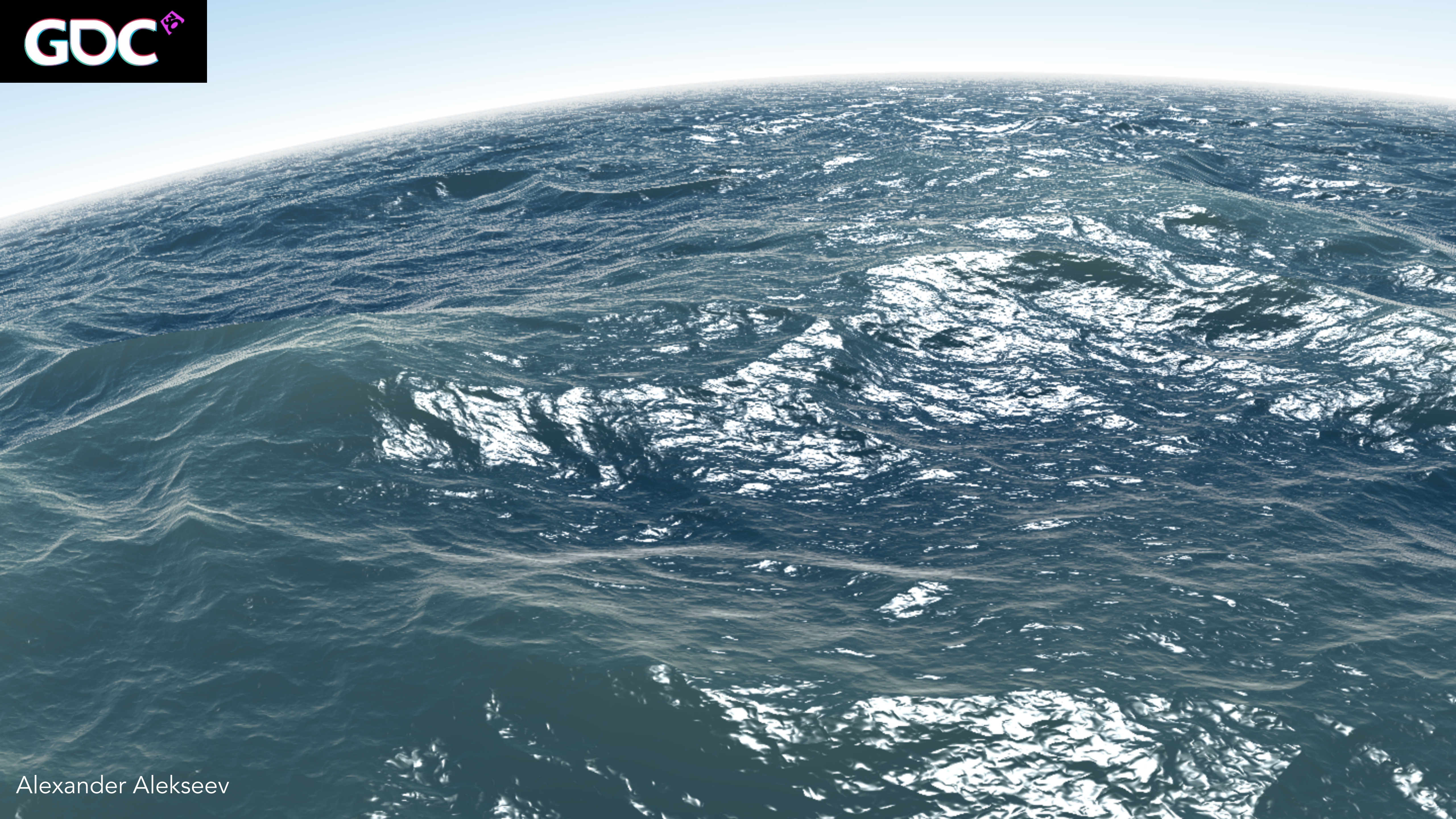
The Map

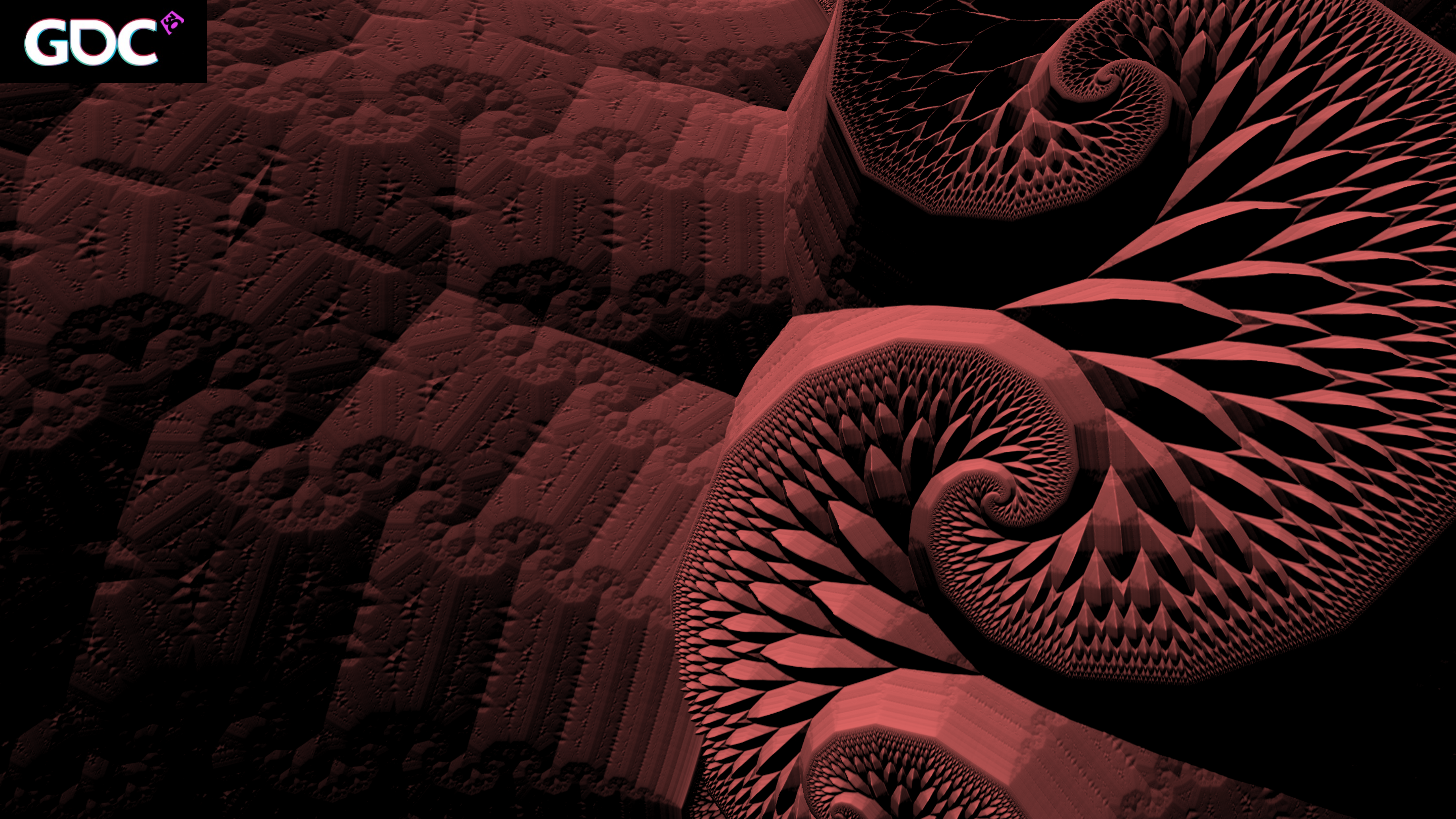


The universe is math.



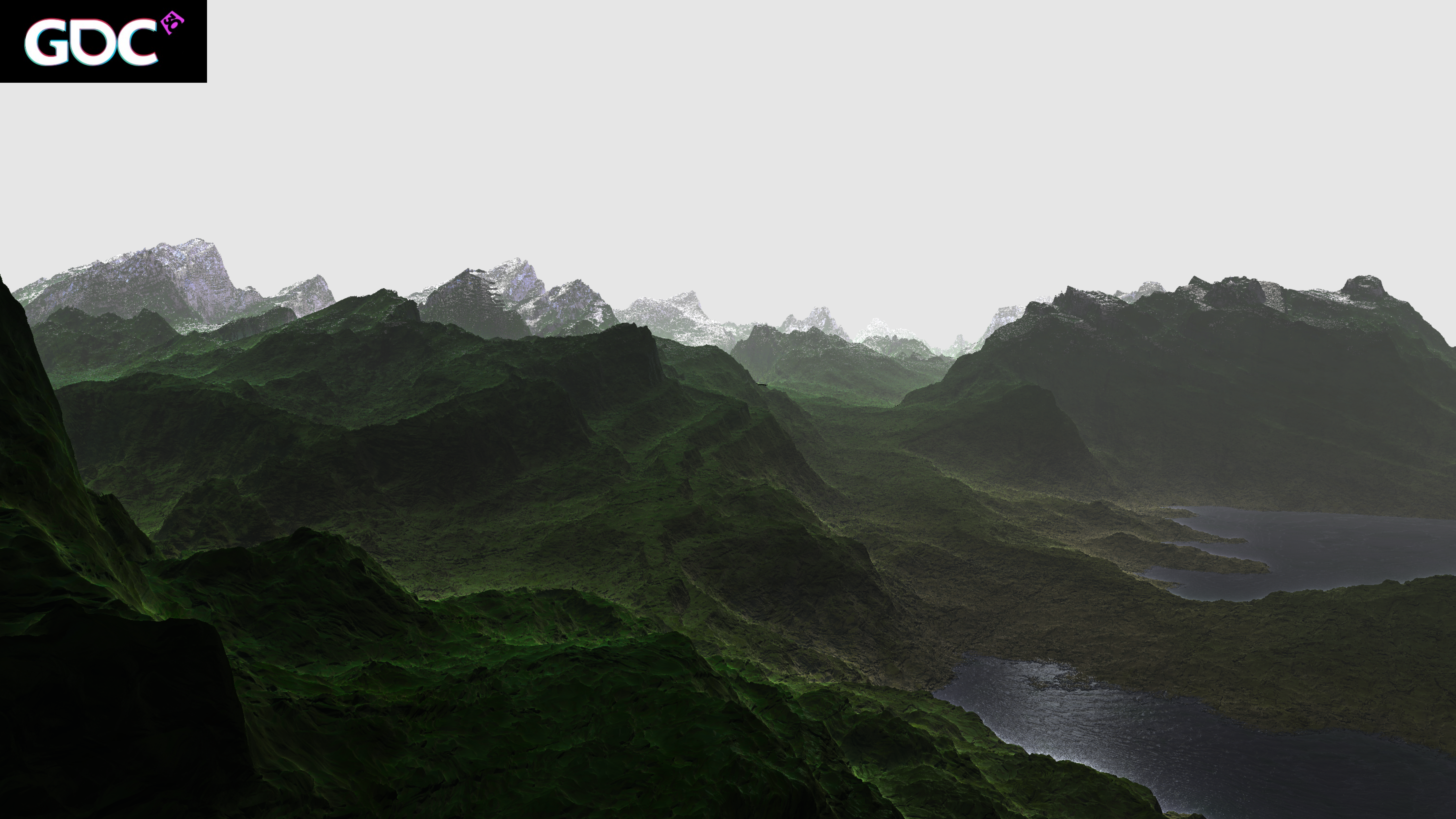




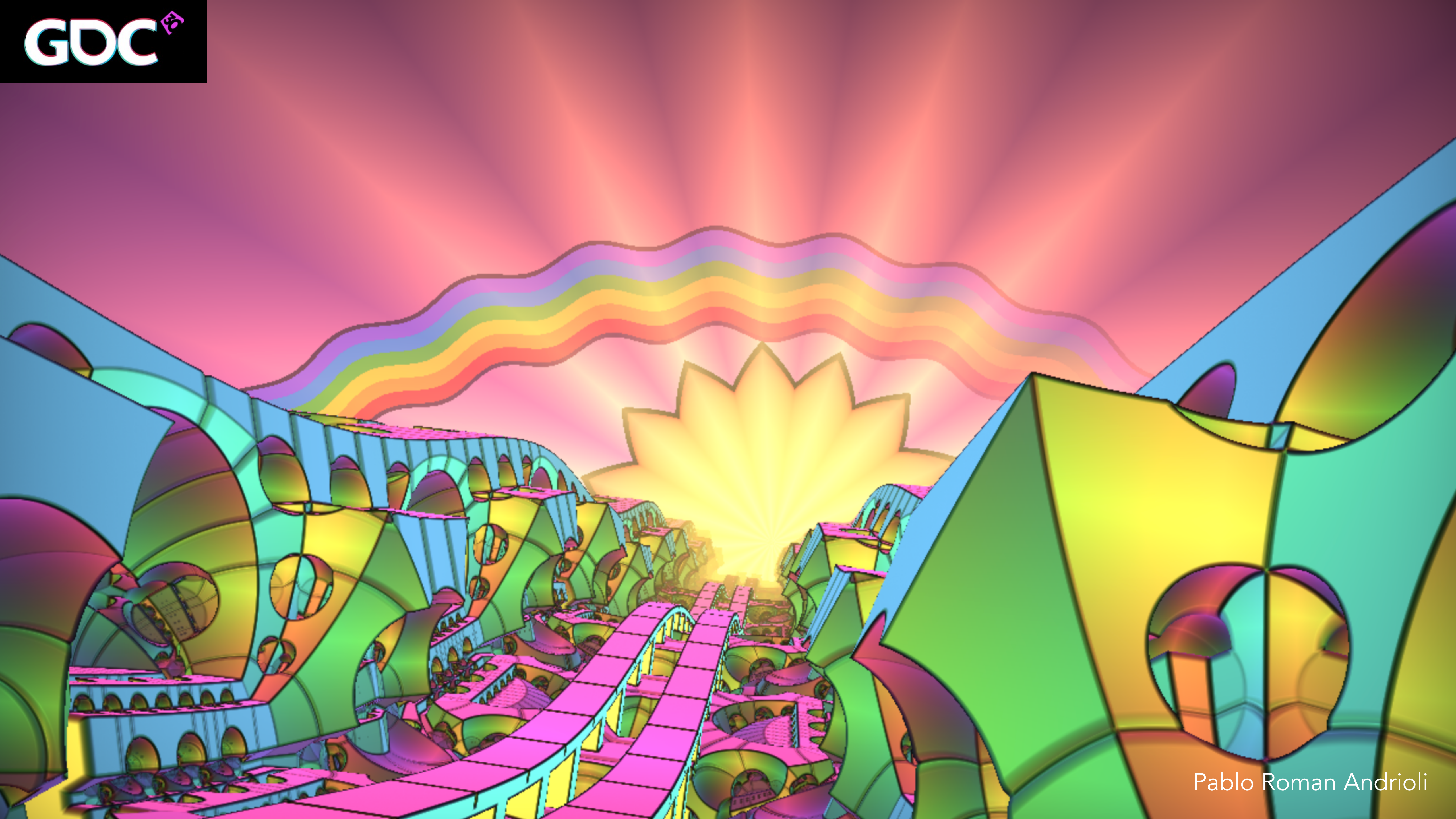


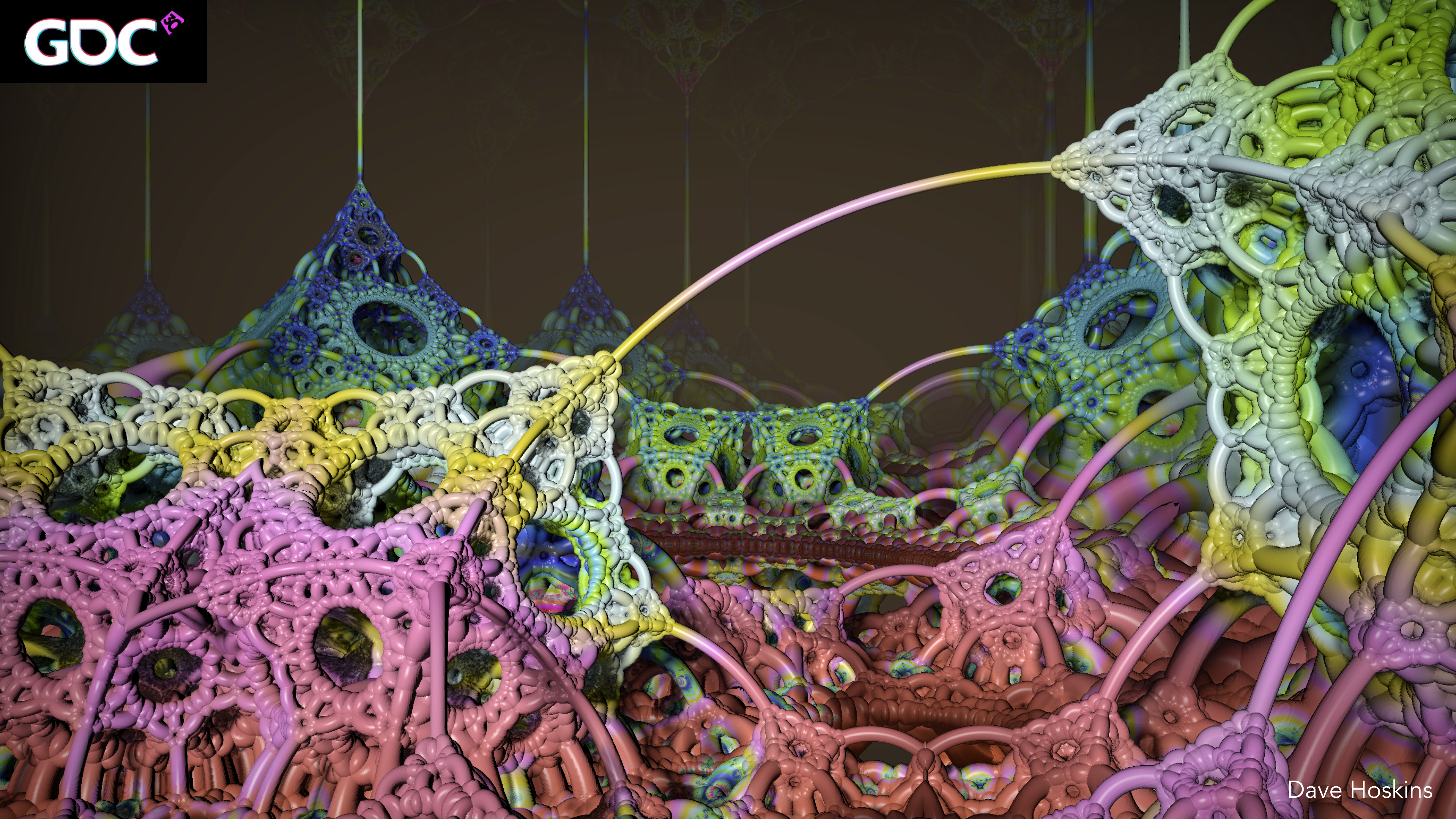


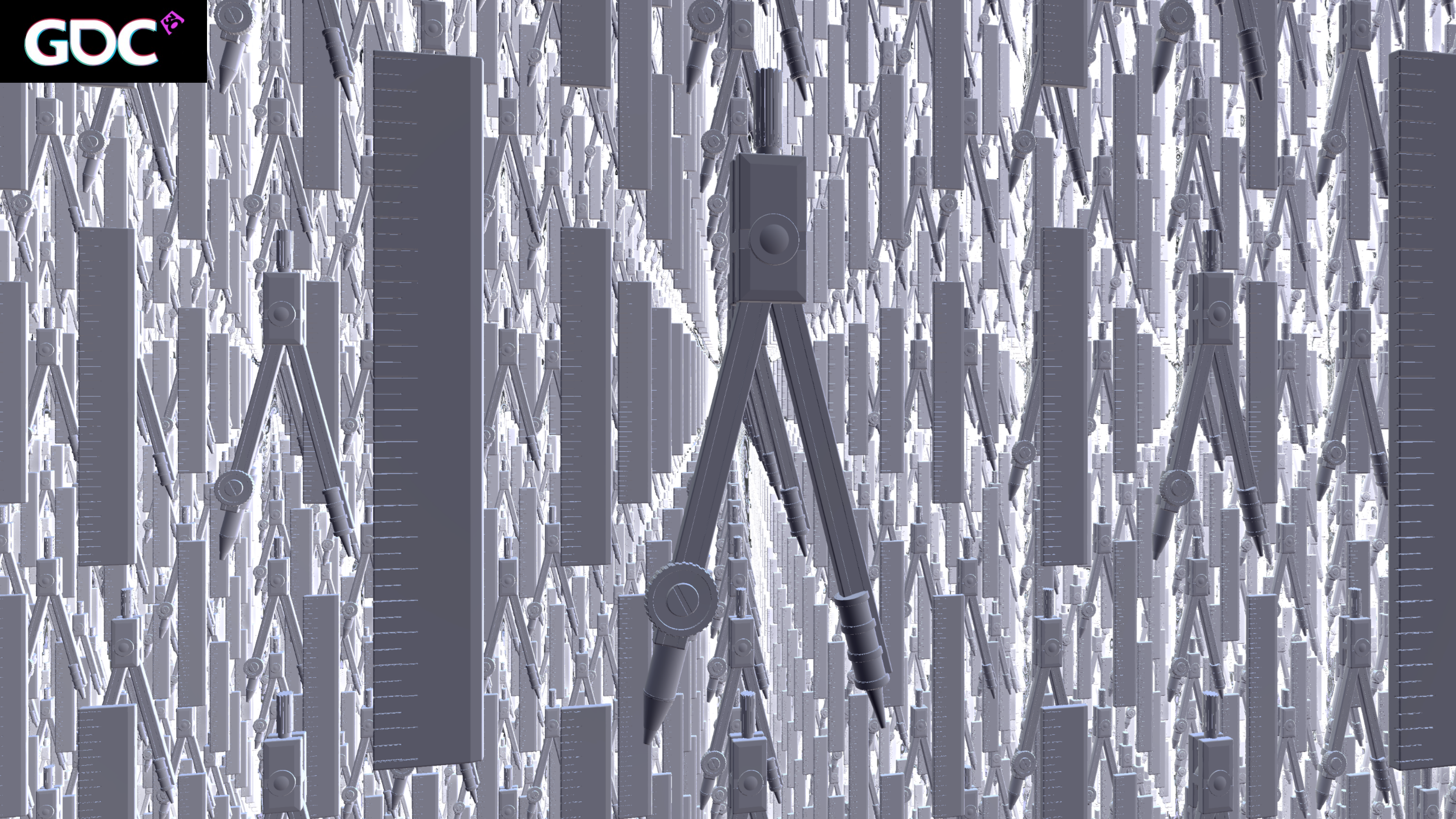


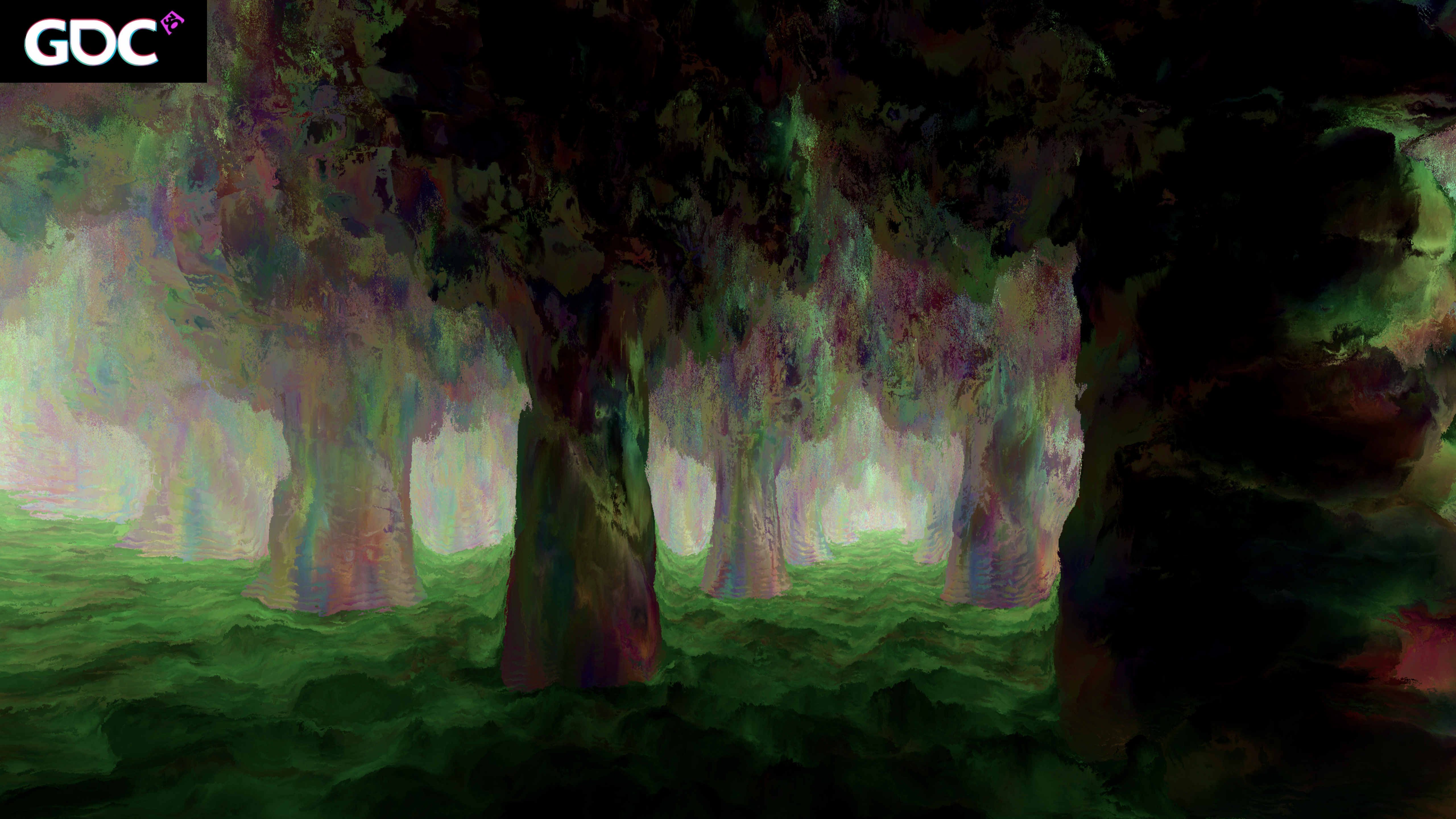














If the universe is math...

Raytracing is a nice way to visit.



Explicit Data vs Implicit Functions



Explicit Data



Explicit Data

Data is stored in memory as independent values.



Explicit Data

Data is stored in memory as independent values.

Mesh vertices, texture pixels, etc...



Explicit Data

Data is stored in memory as independent values.

Mesh vertices, texture pixels, etc...

Data is read from memory.



Implicit Functions



Implicit Functions

Code is data.



Implicit Functions

Code is data.

Everything is procedural.



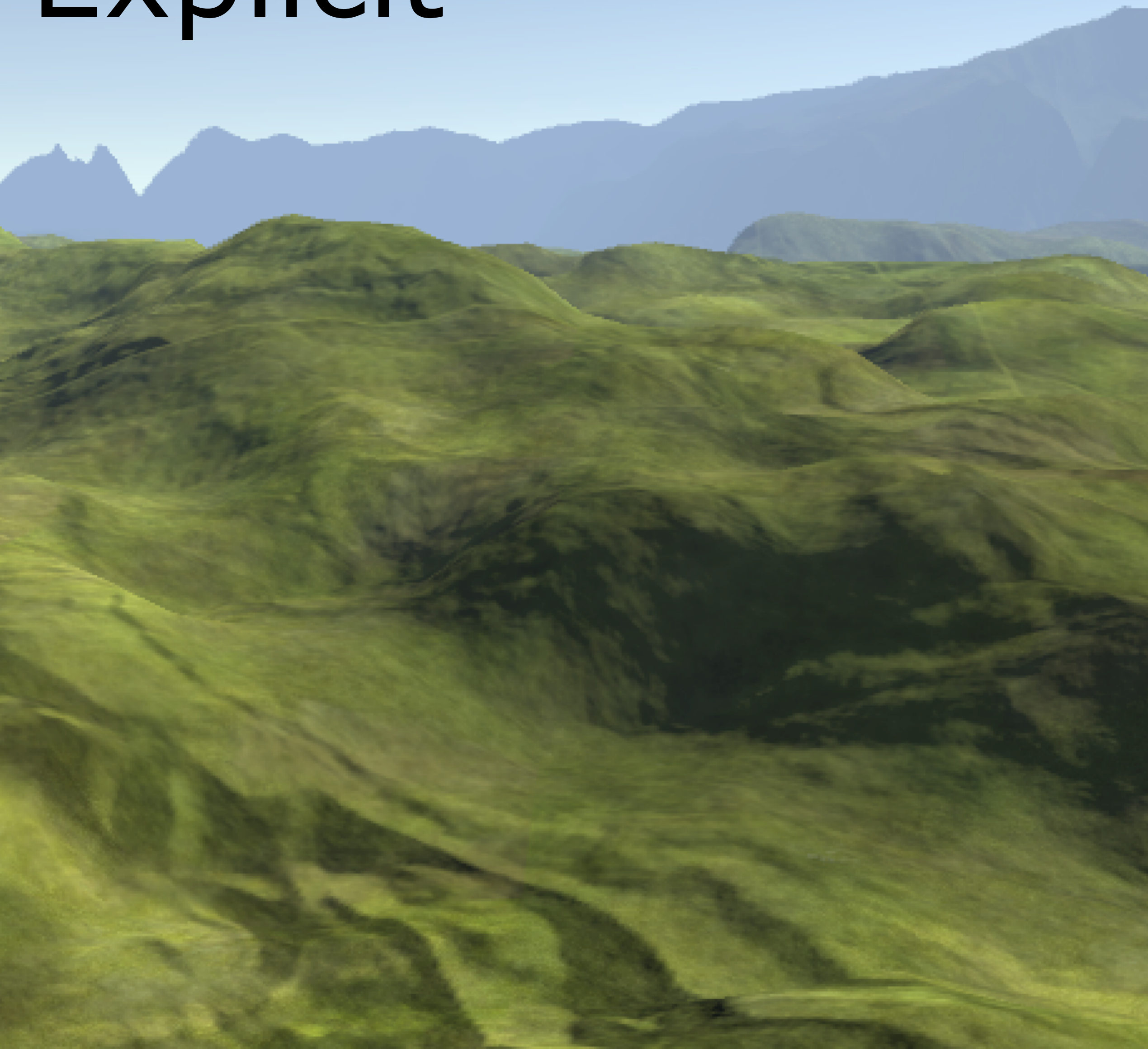
Implicit Functions

Code is data.

Everything is procedural.

Data is accessed via computation.

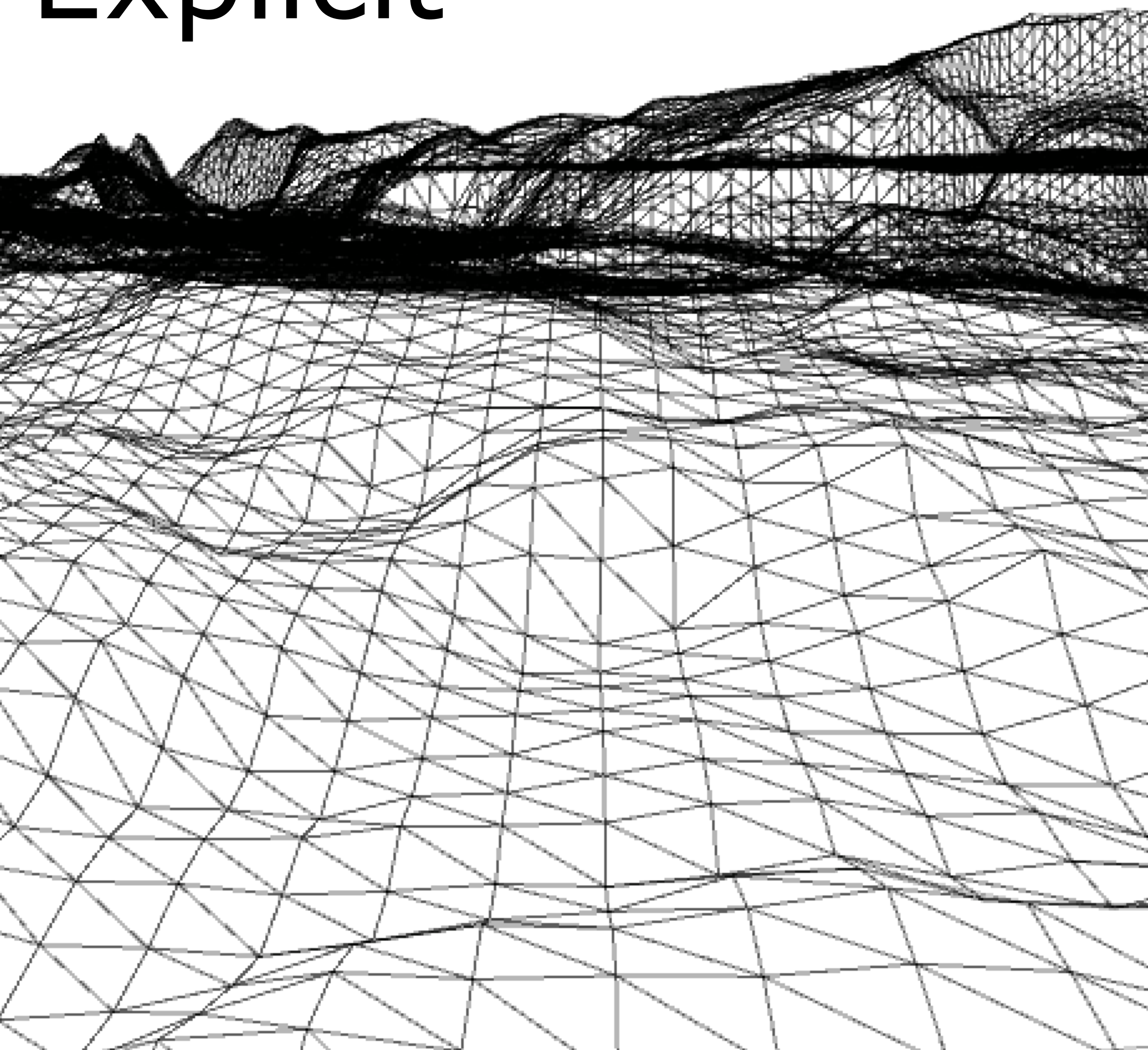
Explicit



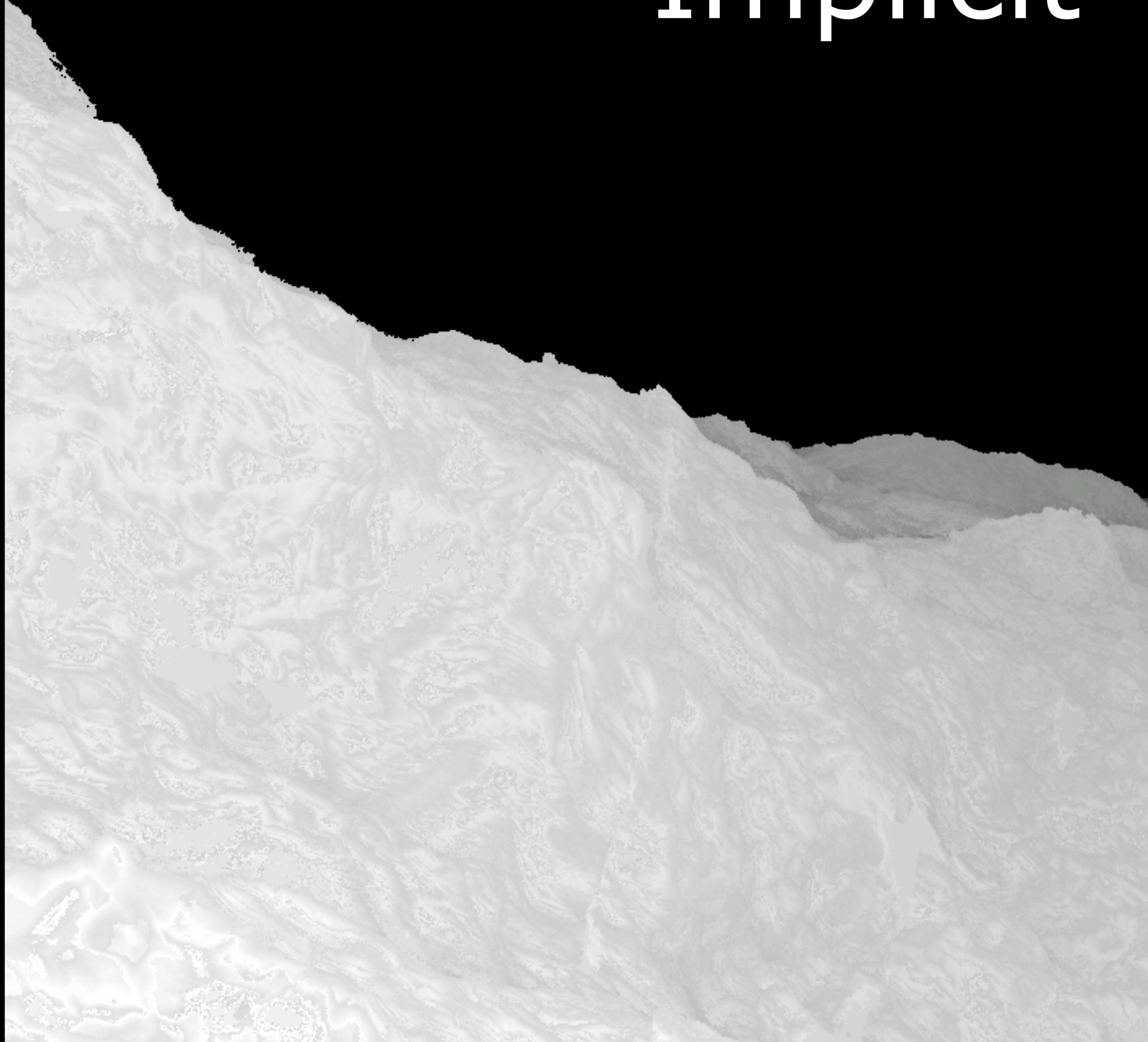
Implicit

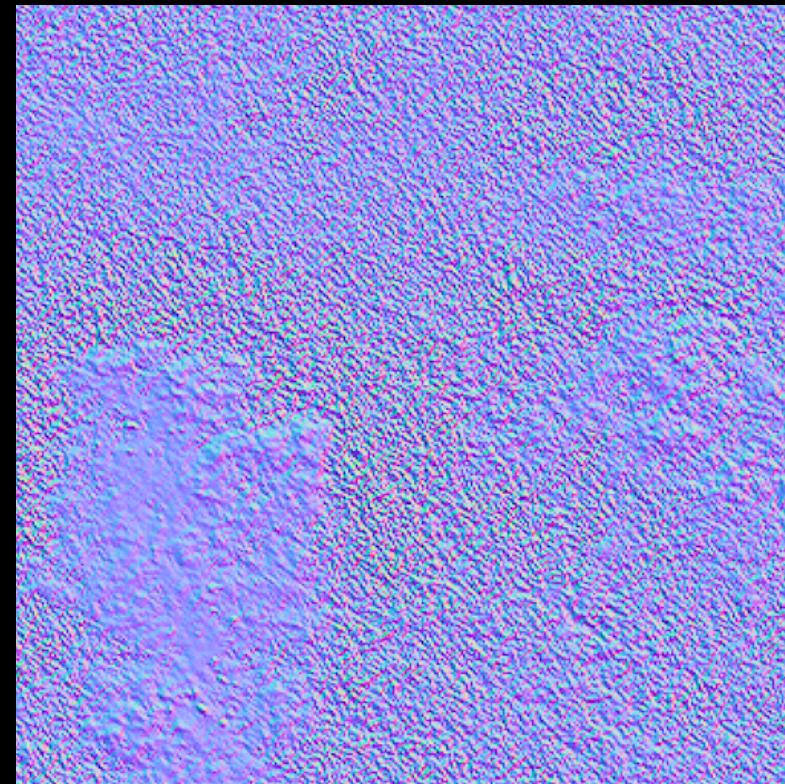
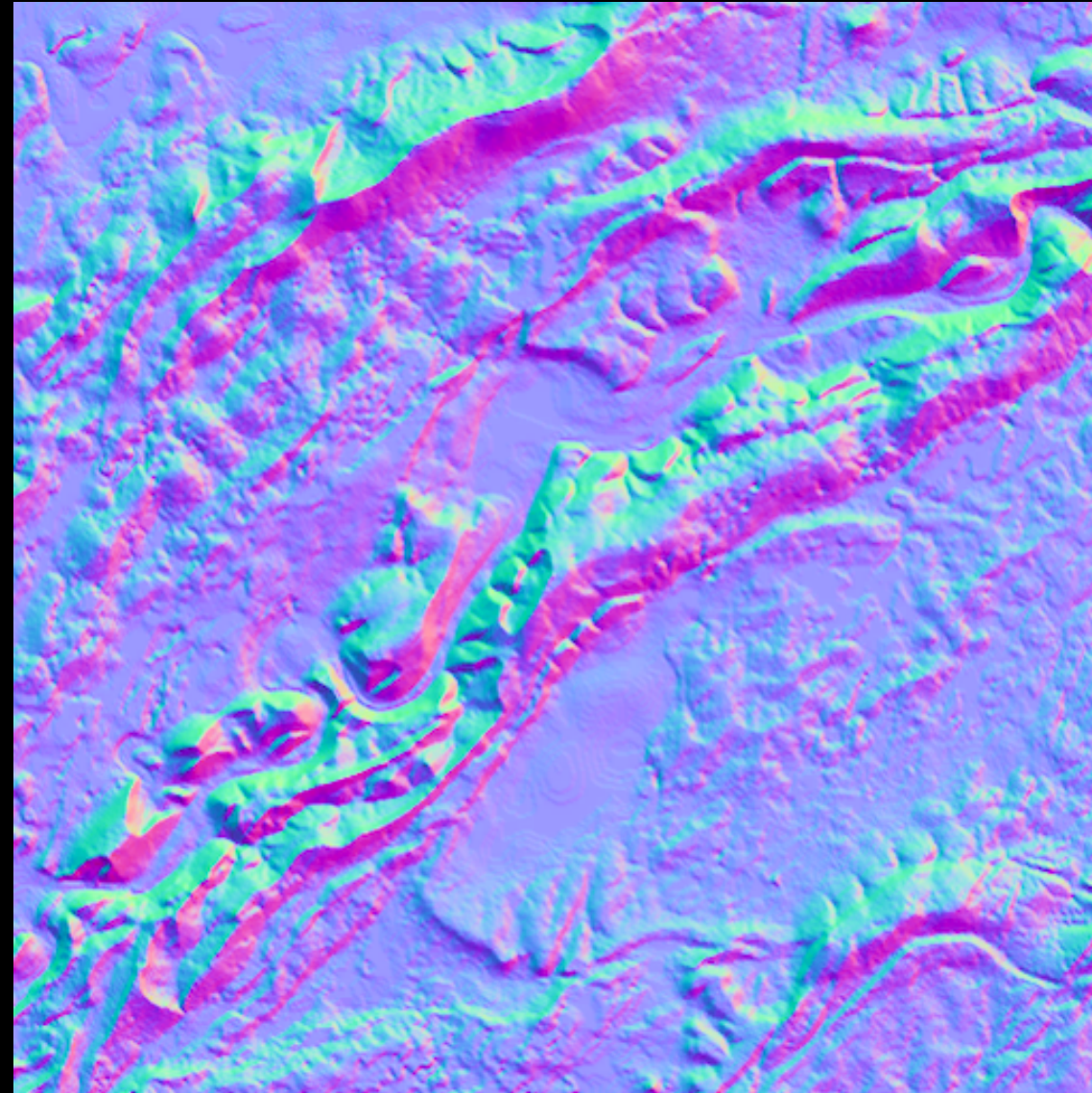
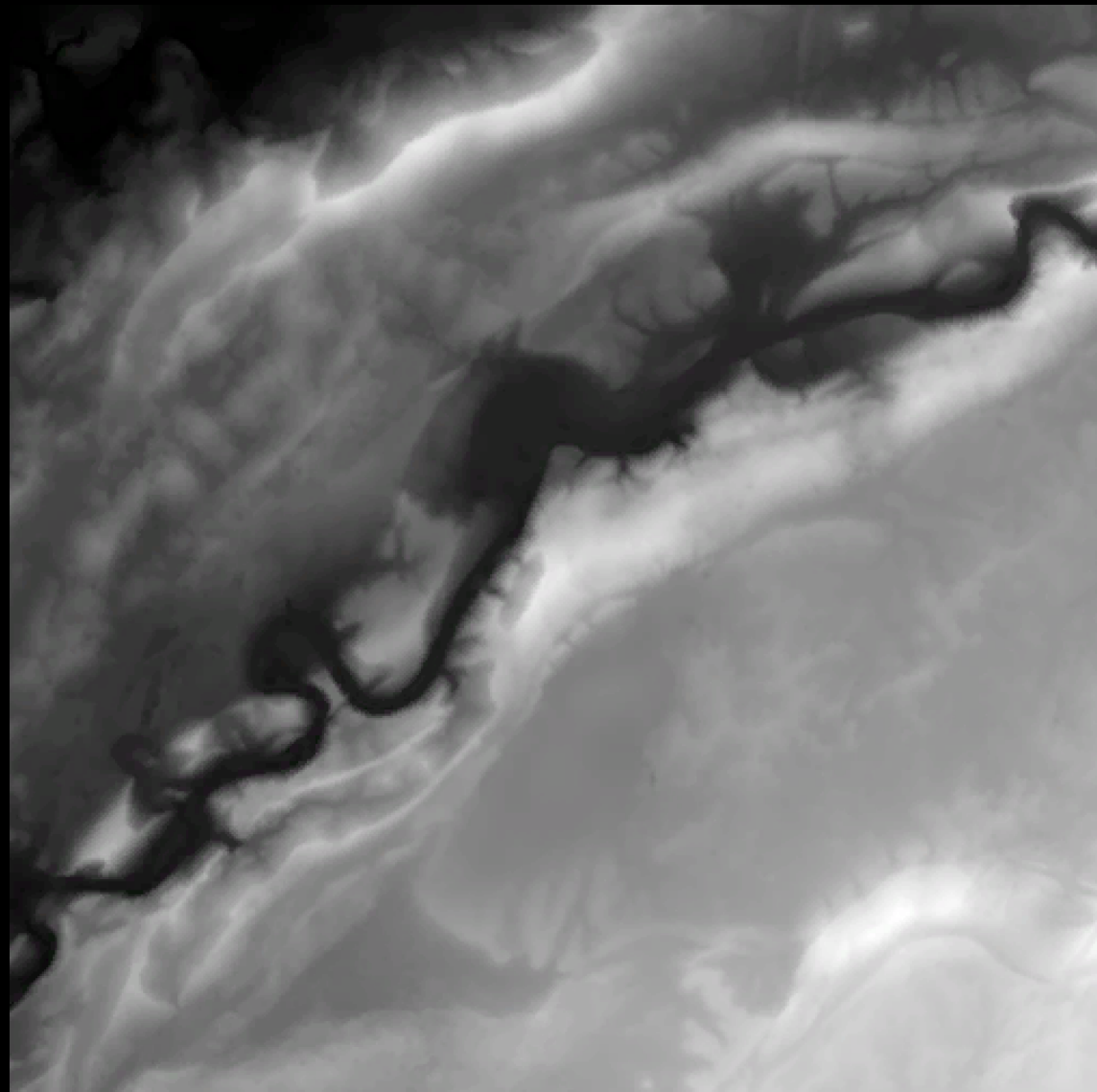


Explicit



Implicit





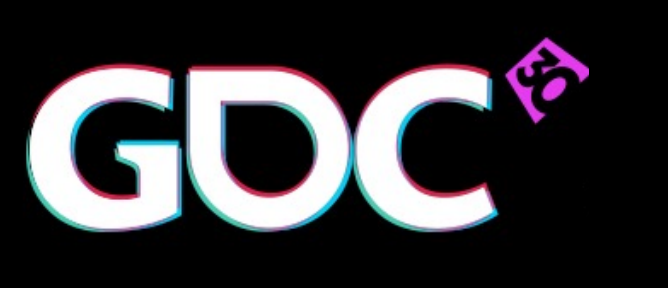
```
float fold(in float x)
{
    return abs(fract(x)-.5);
}

vec3 fold(in vec3 p)
{
    float x = fold(p.x);
    return vec3(fold(p.z+fold(p.y)), fold(p.z+x), fold(p.y+x));
}

float fold_harmonic(in vec3 p, const int it)
{
    vec3 bp = abs(p);

    float rz = 1.;
    for (int i = 0; i < 32; i++)
    {
        if(i>it) break;
        p      *= fold(bp);
        bp     += abs(fract(bp))-rz;
        rz     += dot(fold(p), bp)-rz;

    }
    return rz*float(it);
}
```

Y Plane



```
float y(float height)
{
    return position.y - height;
}
```

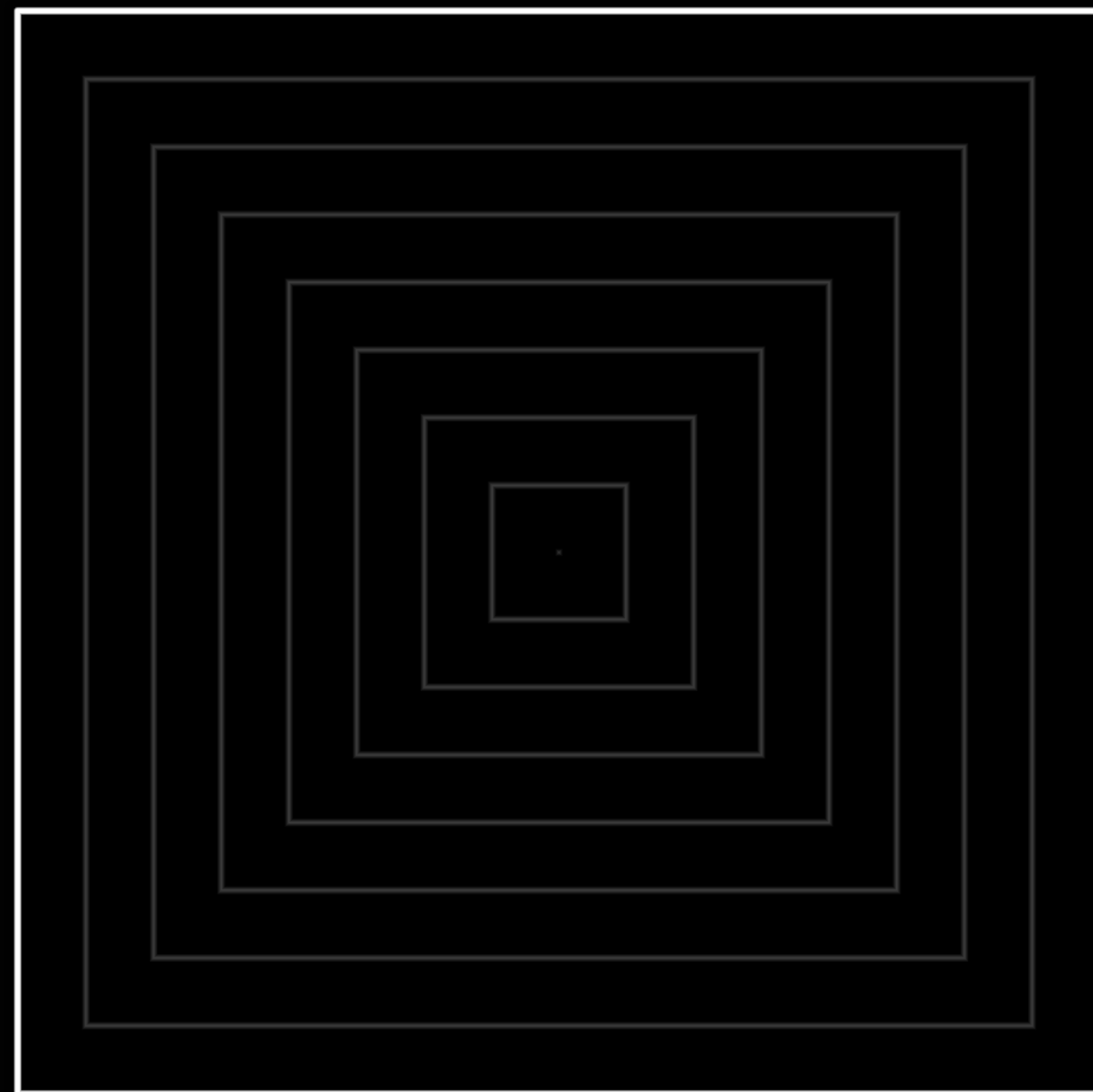

X Plane


```
float x(float offset)
{
    return position.x - offset;
}
```




Circle


```
float sphere(vec2 position, float radius)
{
    return length(position)-radius;
}
```

Square


```
float cube(vec2 position, vec2 scale)
{
    vec2 vertex    = abs(position) - scale;
    vec2 edge      = max(vertex, 0.);
    float interior  = max(vertex.x, vertex.y);

    return min(interior, 0.) + length(edge);
}
```

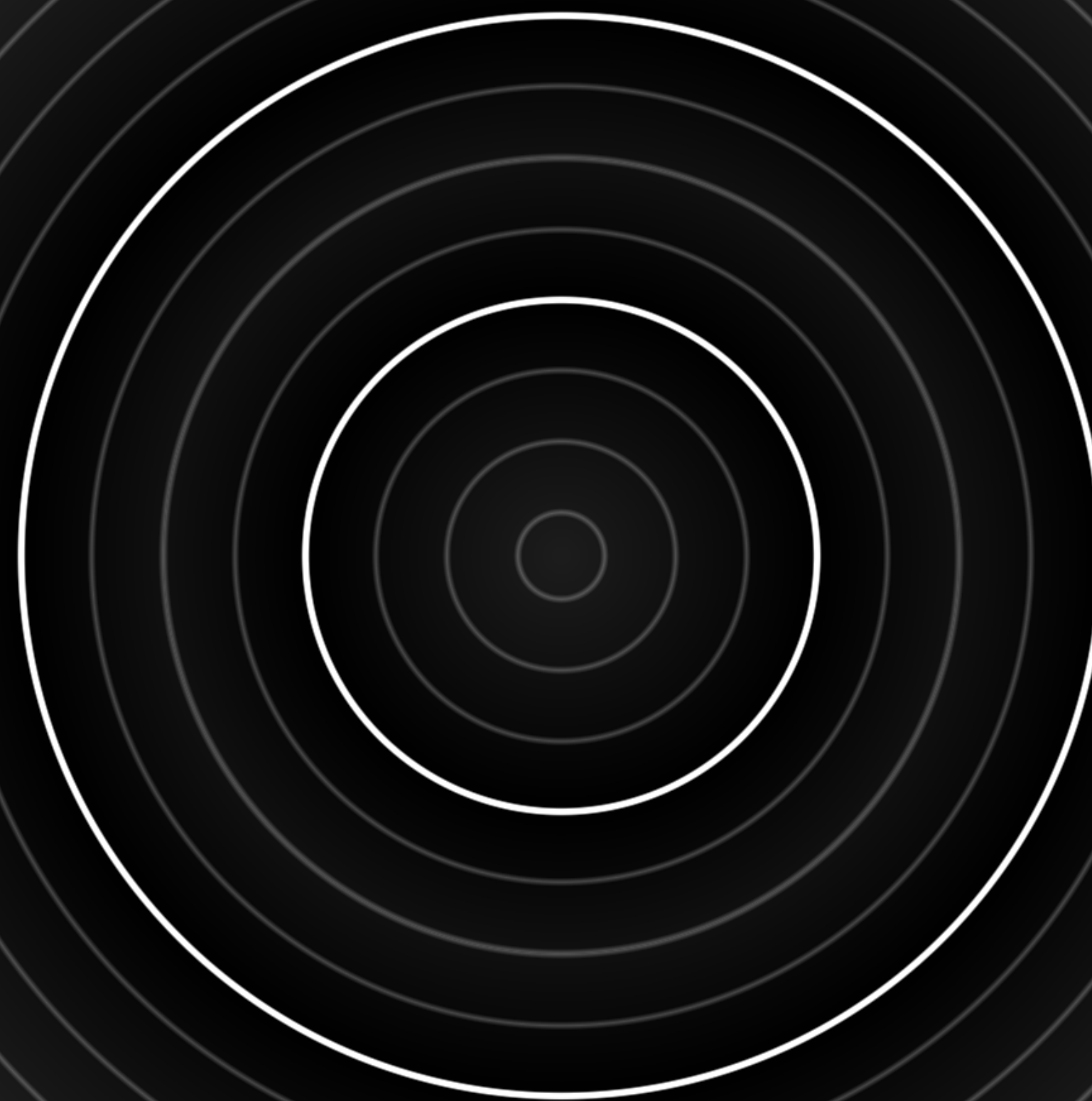



Triangle


```
float prism(vec2 position, float scale)
{
    position.y    *= 0.57735026918; //1./sqrt(3.);
    scale         *= 0.57735026918;

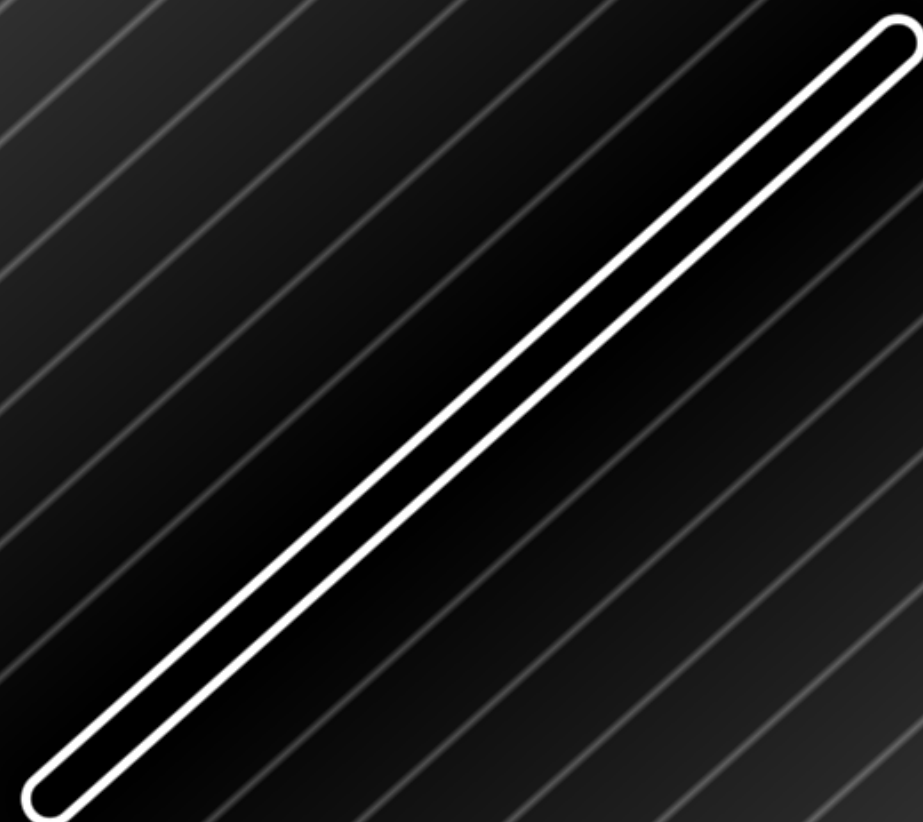
    vec3 edge     = vec3(0.);
    edge.x        = position.y + position.x;
    edge.y        = position.x - position.y;
    edge.z        = position.y + position.y;
    edge          *= .86602540358; //sqrt(3.)/2.;

    return max(edge.x, max(-edge.y, -edge.z))-scale;
}
```

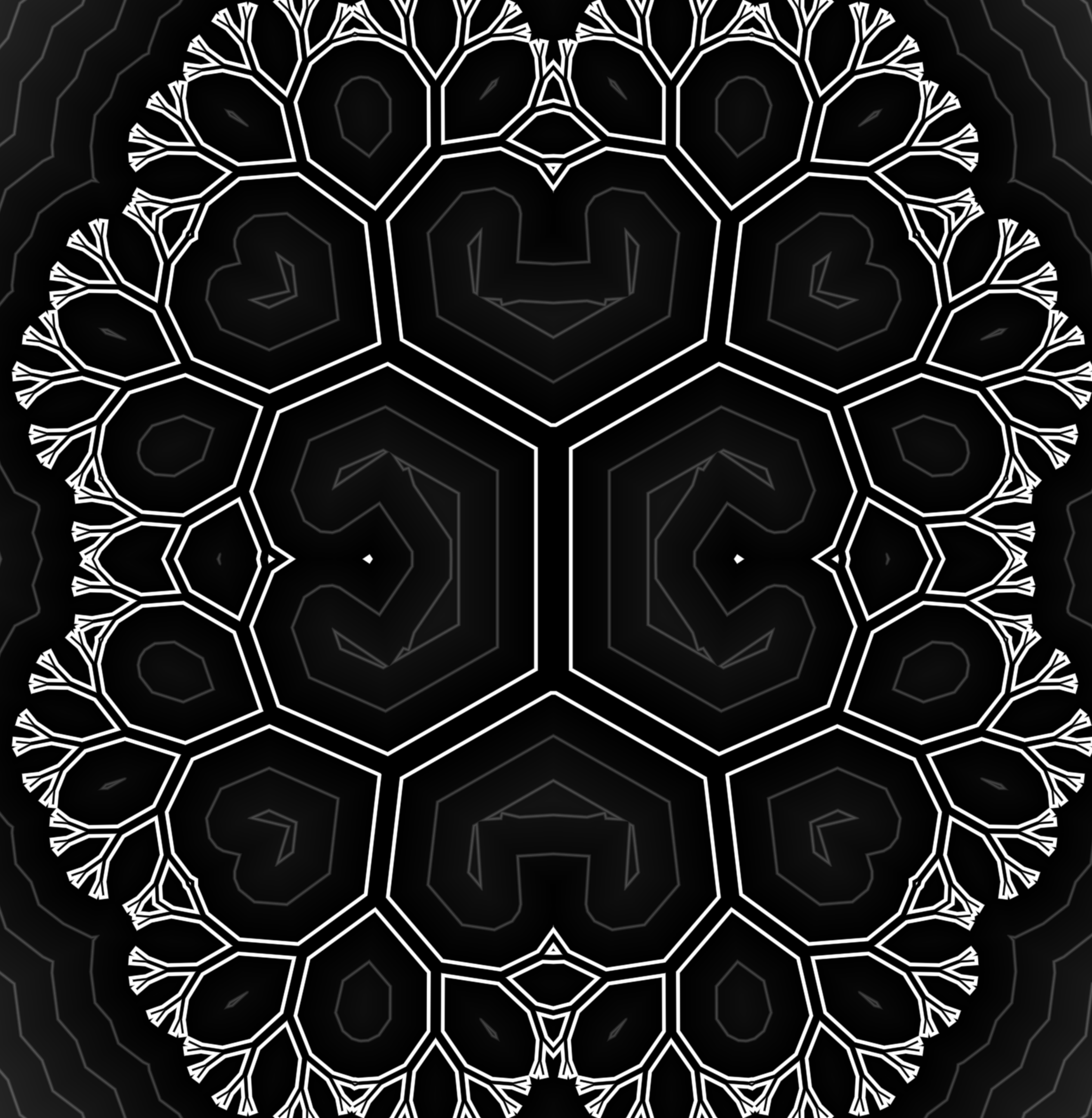
Torus


```
float torus(vec2 position, vec2 radius)
{
    return abs(abs(length(position)-radius.x)-radius.y);
}
```

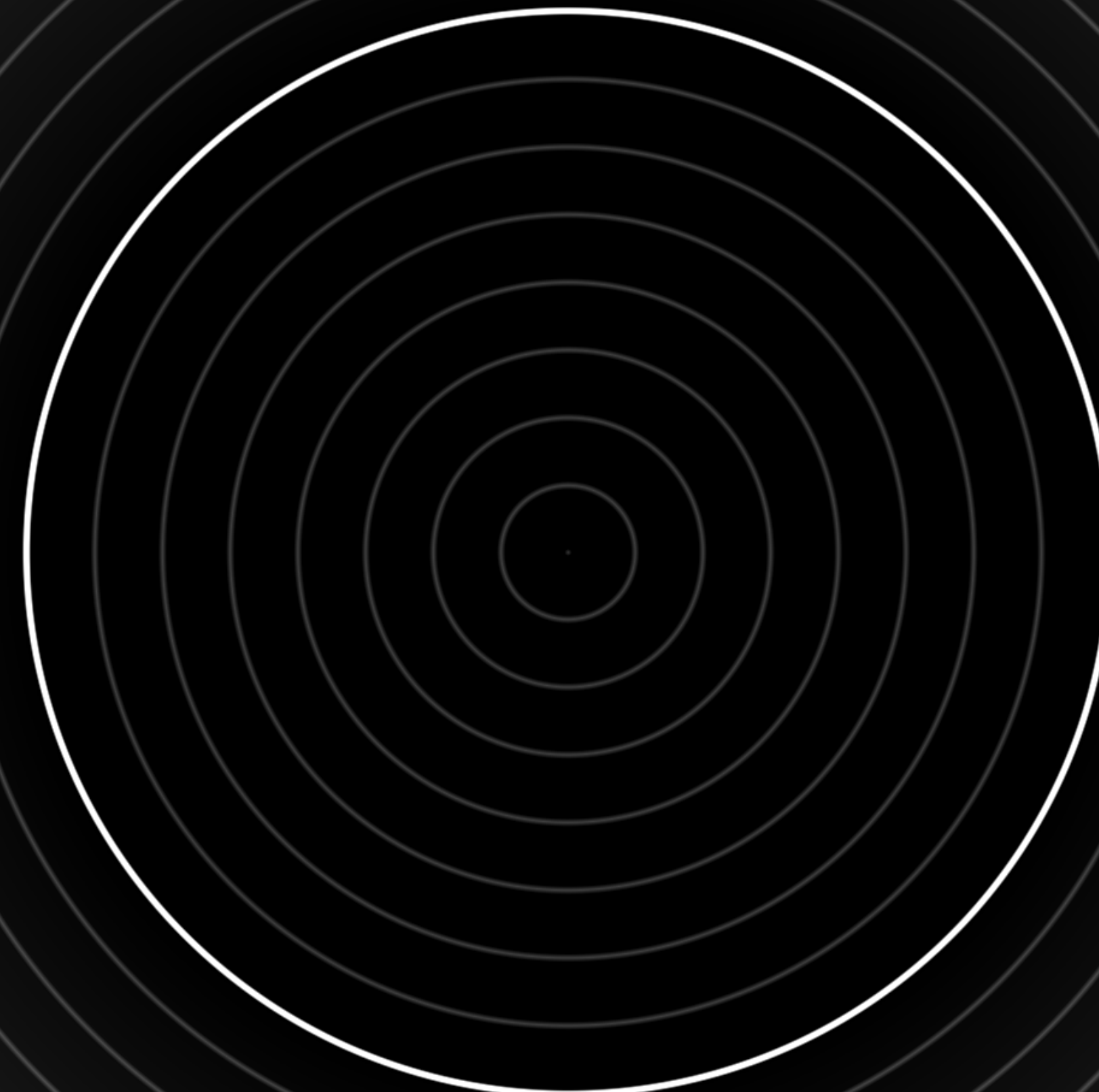
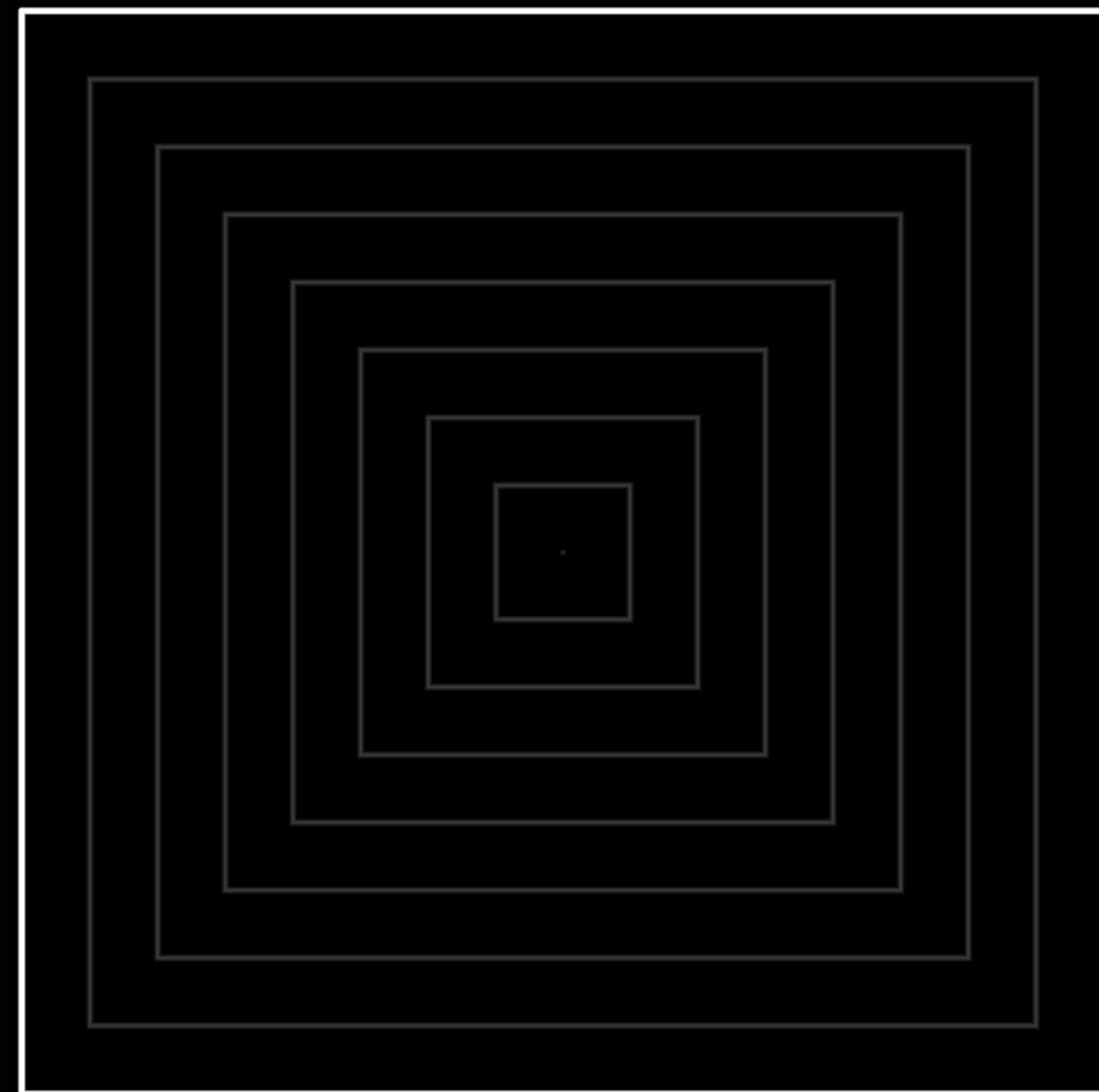
Segment


```
float torus(vec2 position, vec2 radius)
{
    return abs(abs(length(position)-radius.x)-radius.y);
}
```

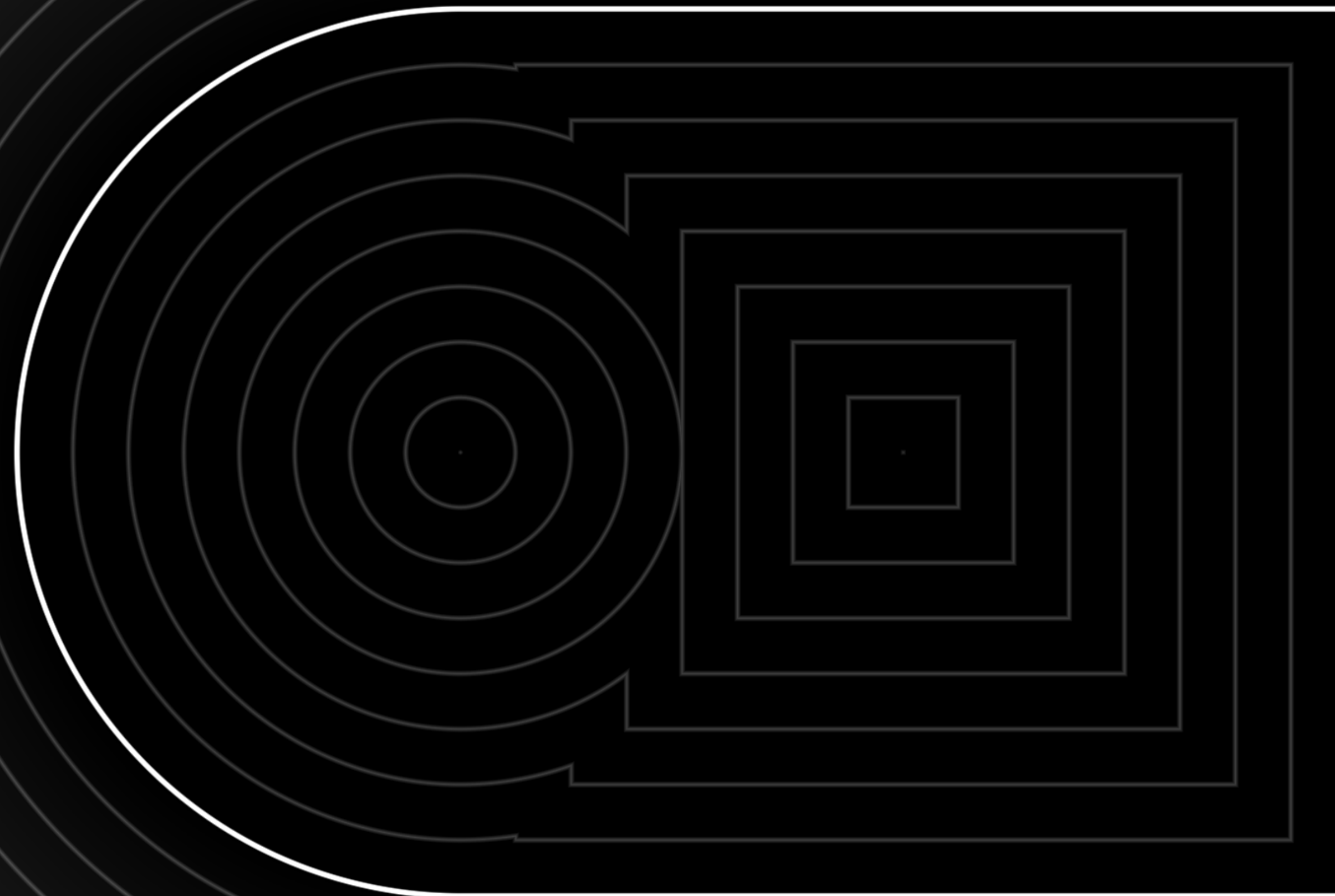
Fractal


```
float fractal(vec2 position, float rotation, float translation, float scale)
{
    const int iterations      = 8;
    float radius              = 1.5/float(iterations);
    rotation                  = rotation * TAU + TAU * .5;
    position                  = abs(position);
    position.y                -= radius;
    scale                     *= radius;
    float result              = max(position.x, position.y) - scale;
    for (int i = 0; i < iterations; i++)
    {
        float magnitude       = length(position);
        float phase           = atan(position.x, position.y);
        phase                 = mod(phase, rotation) - rotation * .5;
        position.x            = magnitude * cos(phase);
        position.y            = magnitude * sin(phase);
        position               = abs(position);
        position.y            -= translation;
        translation            *= .7;
        scale                  *= .8;
        rotation               *= -.98;
        result                 = min(max(position.x, position.y) - scale, result);
    }
    return result;
}
```

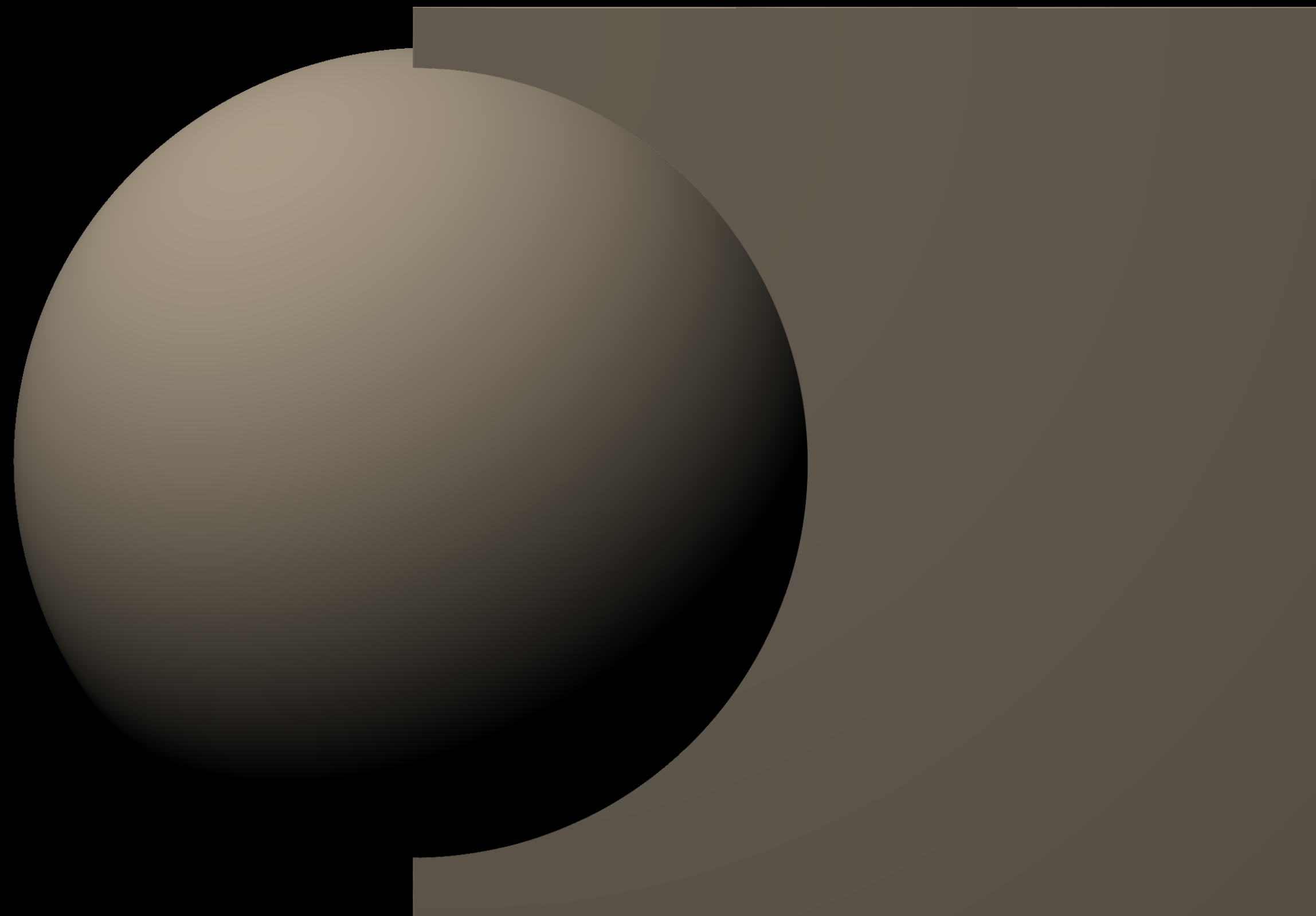
Minimum

`min(a, b)`



Minimum

`min(a, b)`

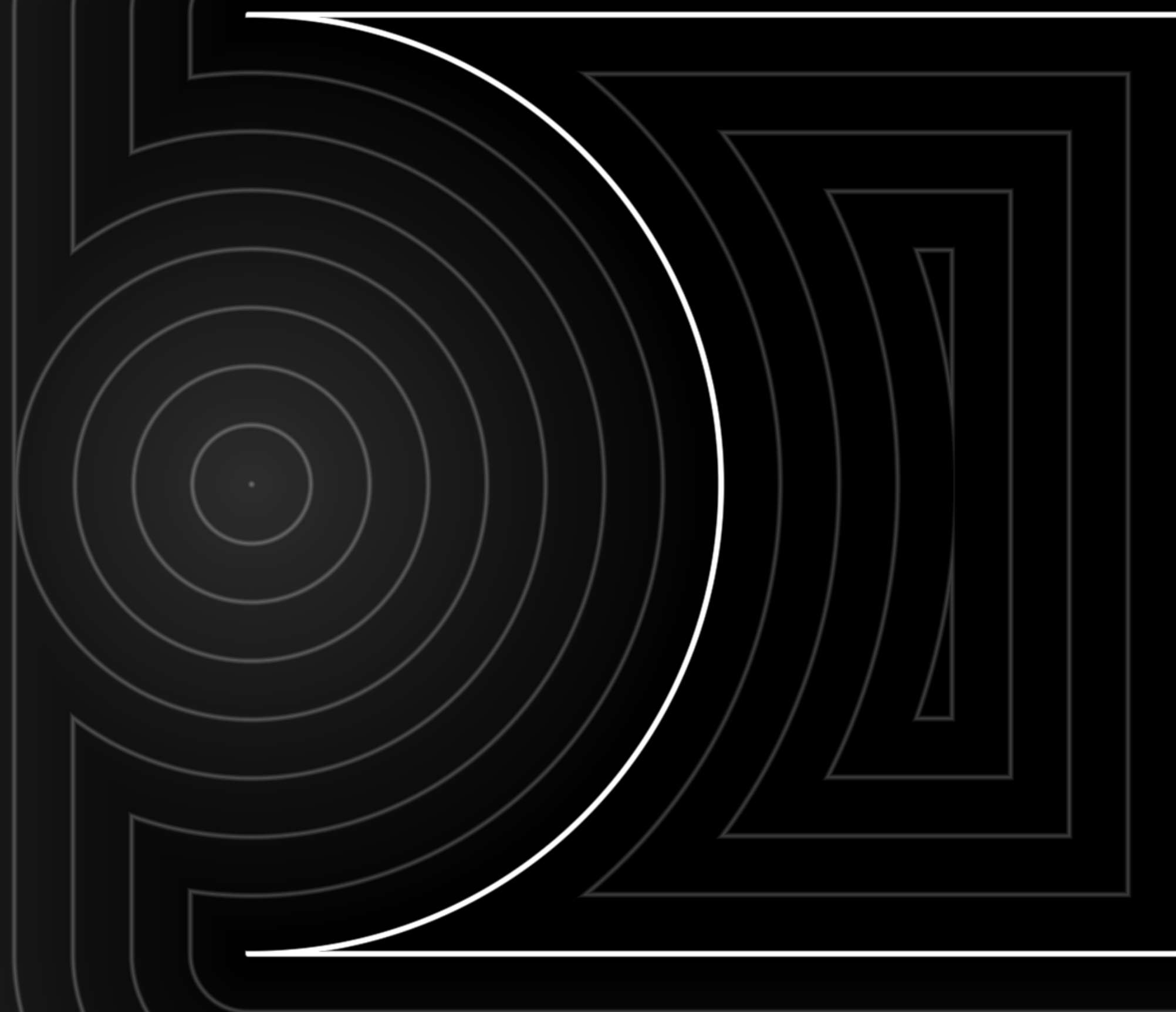




Maximum

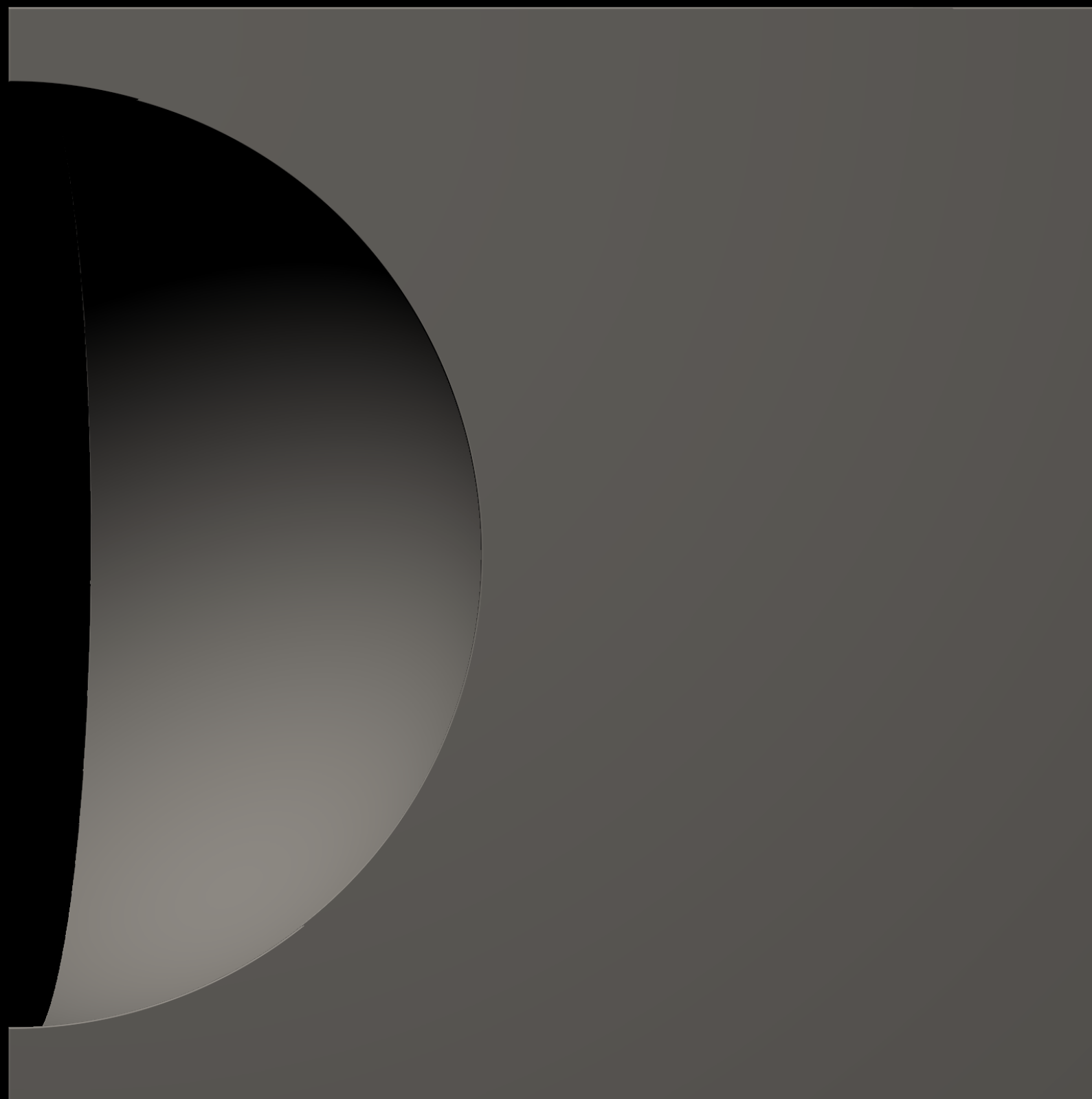
`max(a, b)`





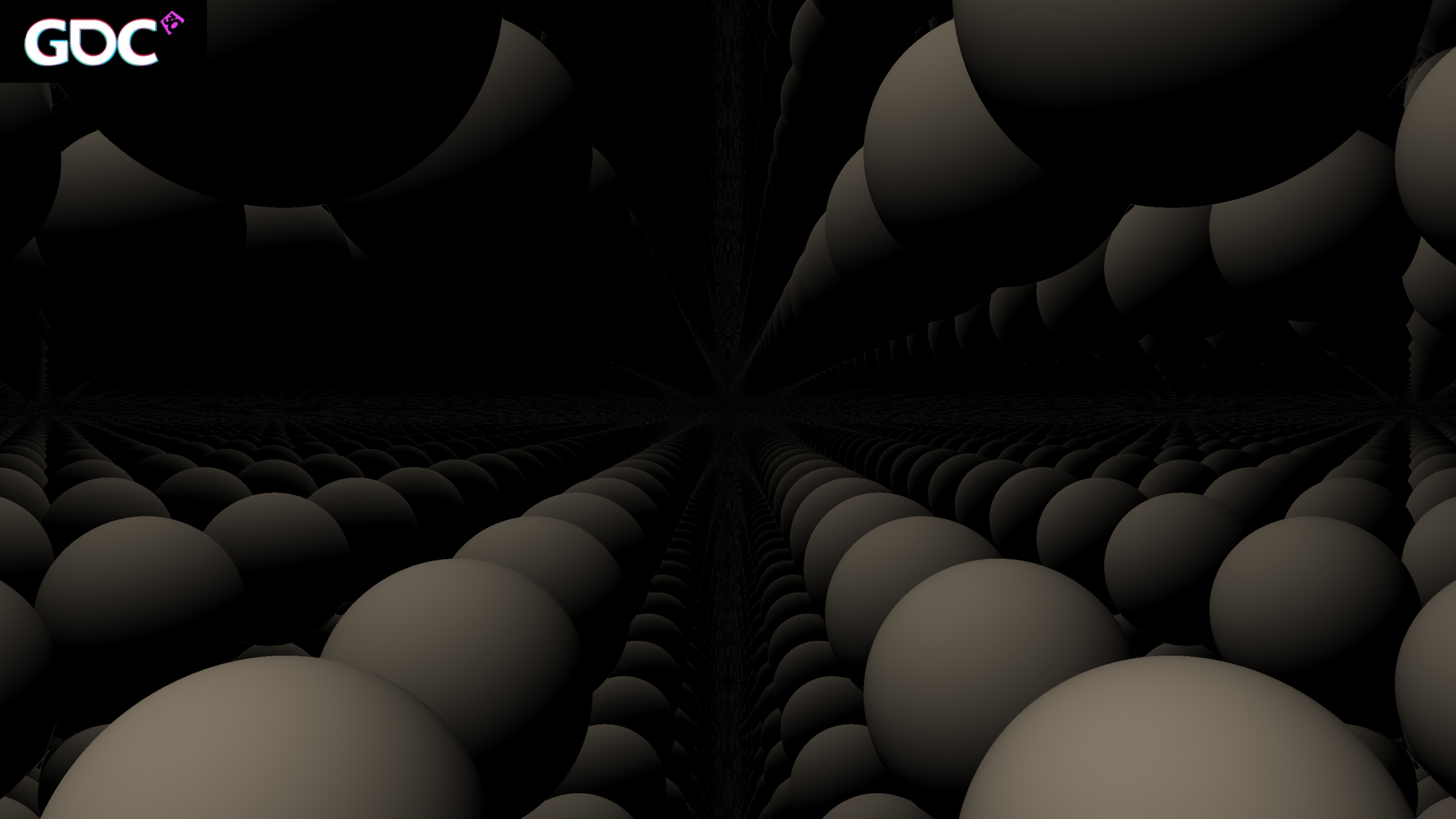
Subtraction

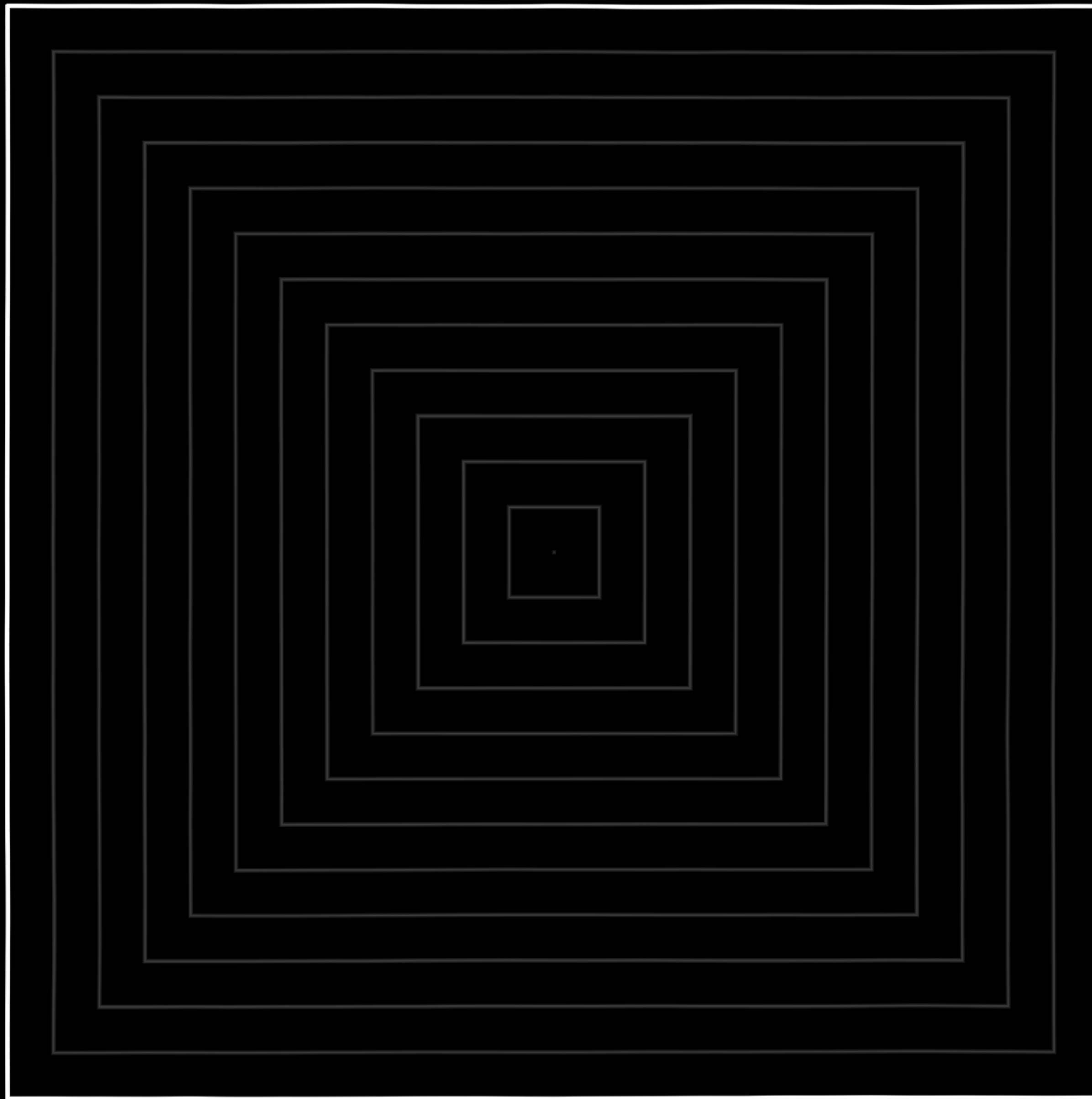

$$\max(-a, b)$$

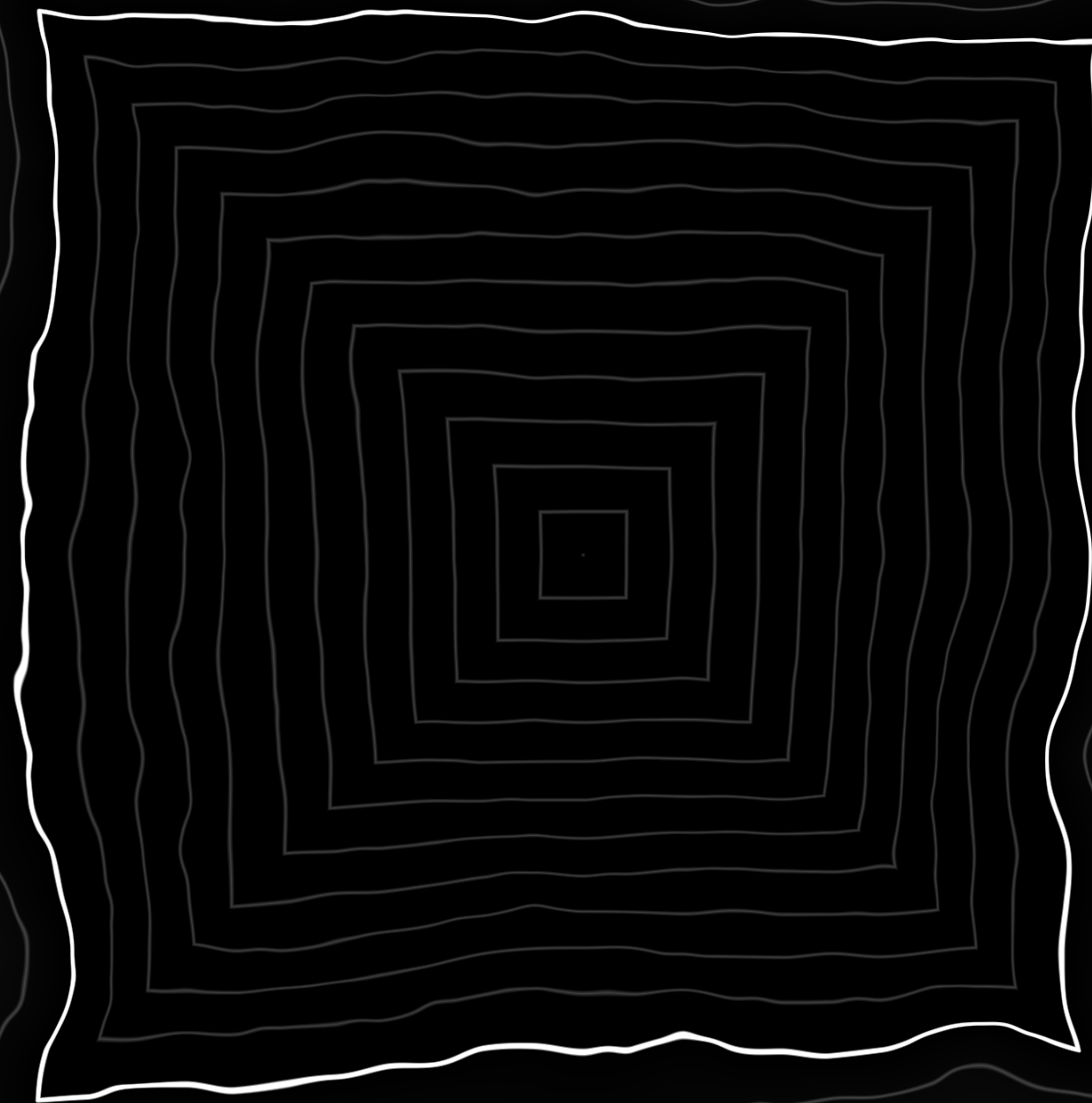


Modulus


```
mod(position, scale) - scale * .5
```





Noise


```
float noise(in vec2 uv)
{
    const float k      = 257.;
    vec4 l             = vec4(floor(uv),fract(uv));
    l.zw               = l.zw*l.zw*(3.-2.*l.zw);

    float u            = l.x + l.y * k;

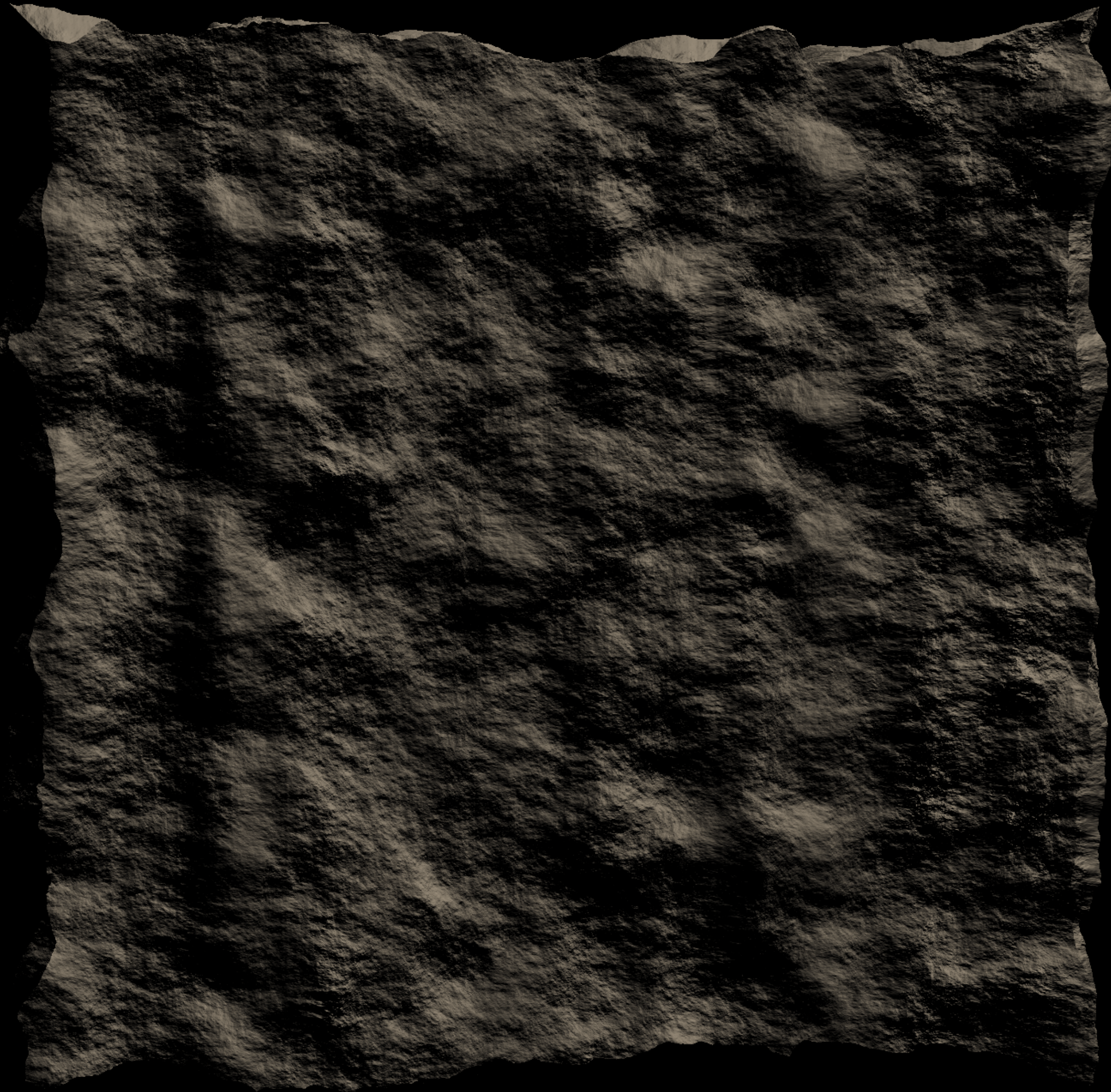
    vec4 v             = vec4(u, u+1.,u+k, u+k+1.);
    v                  = fract(fract(1.23456789*v)*v/.987654321);

    l.x                = mix(v.x, v.y, l.z);
    l.y                = mix(v.z, v.w, l.z);
    return mix(l.x, l.y, l.w);
}

float fractal_brownian_motion(vec2 uv)
{
    float amplitude = .5;
    float frequency = 2.;

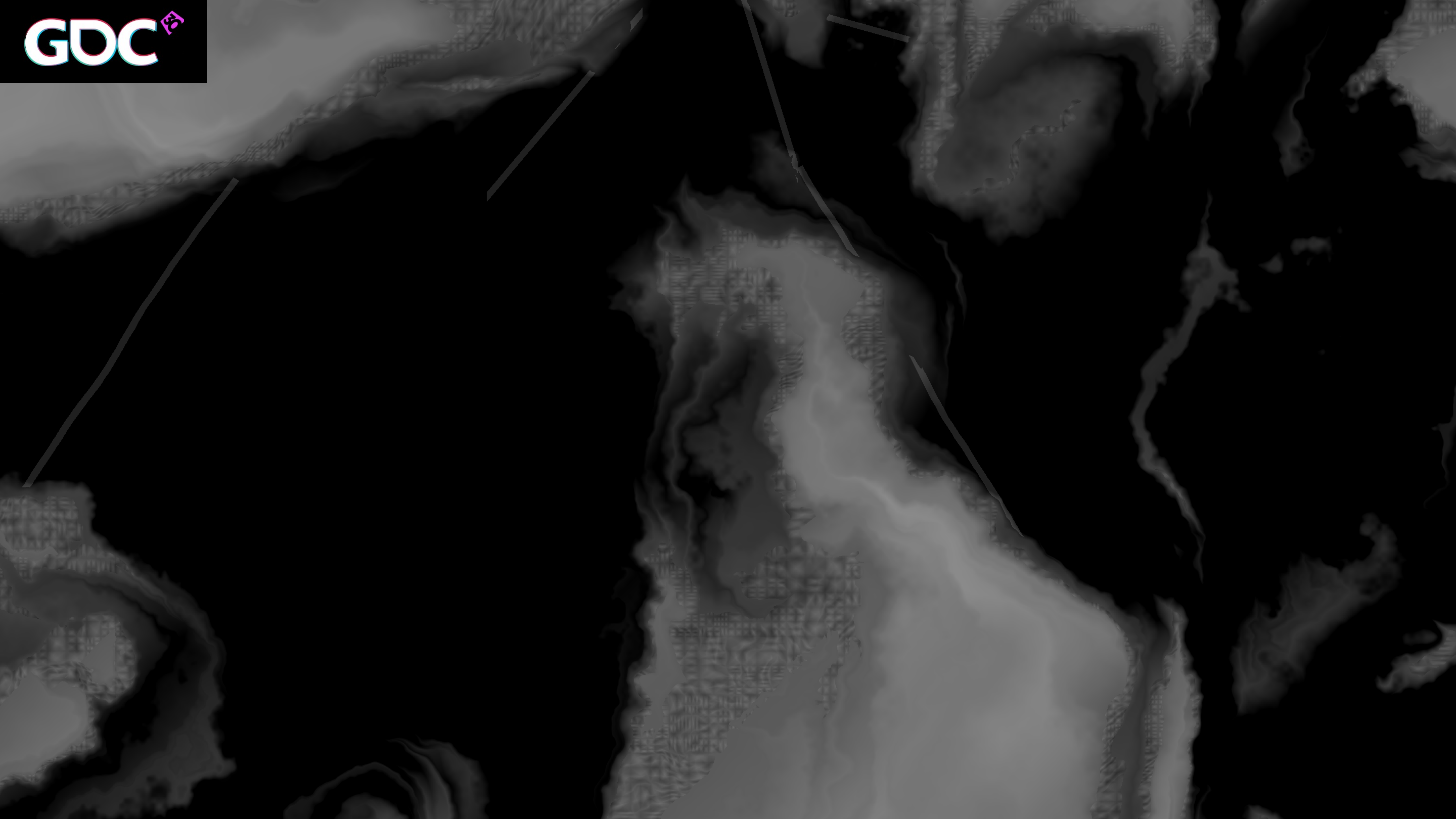
    float result = 0.;
    for(int i = 0; i < 8; i++)
    {
        result      += noise(uv*frequency)*amplitude;
        amplitude    *= .5;
        frequency    *= 2.;
    }

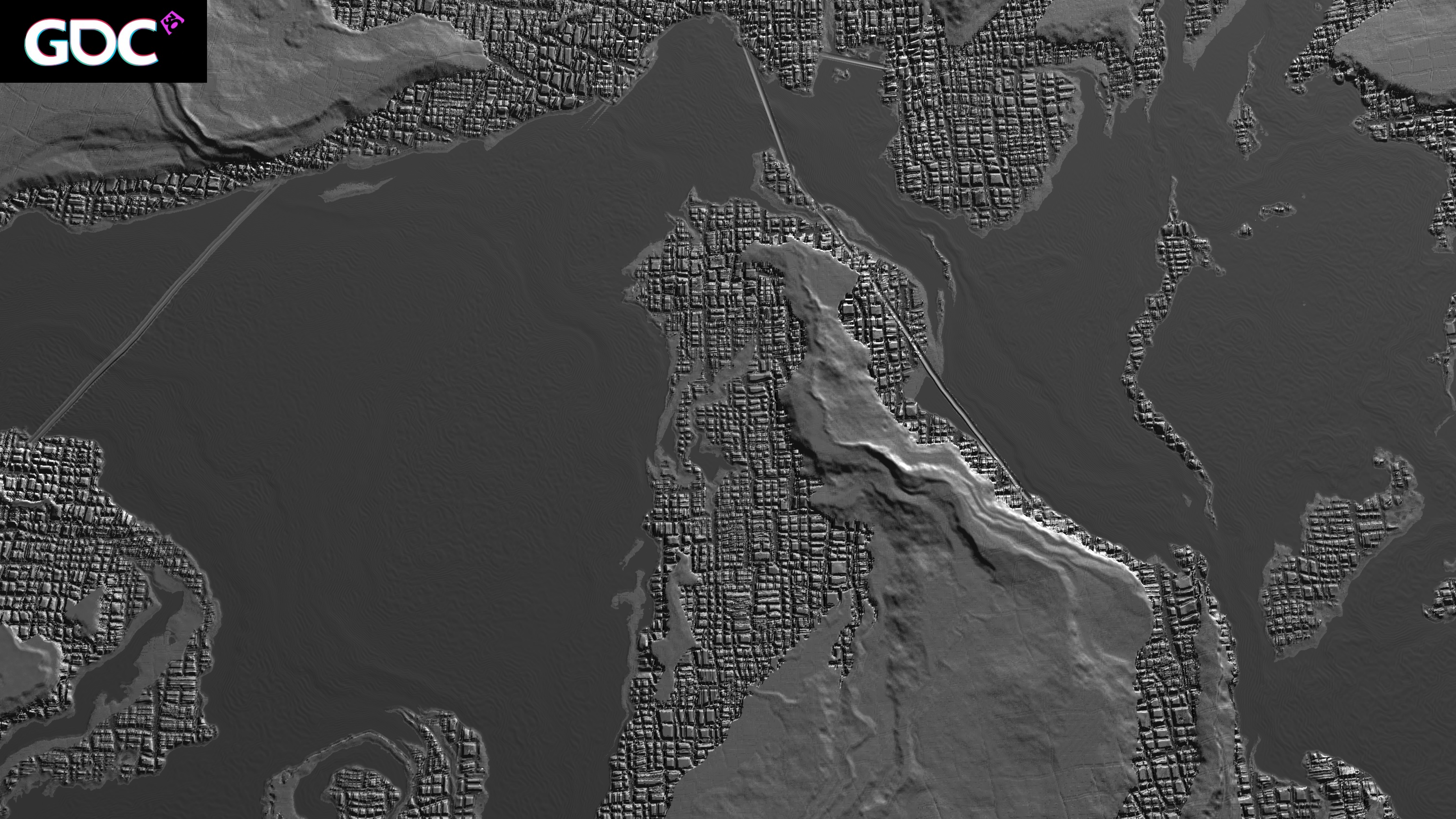
    return result;
}
```

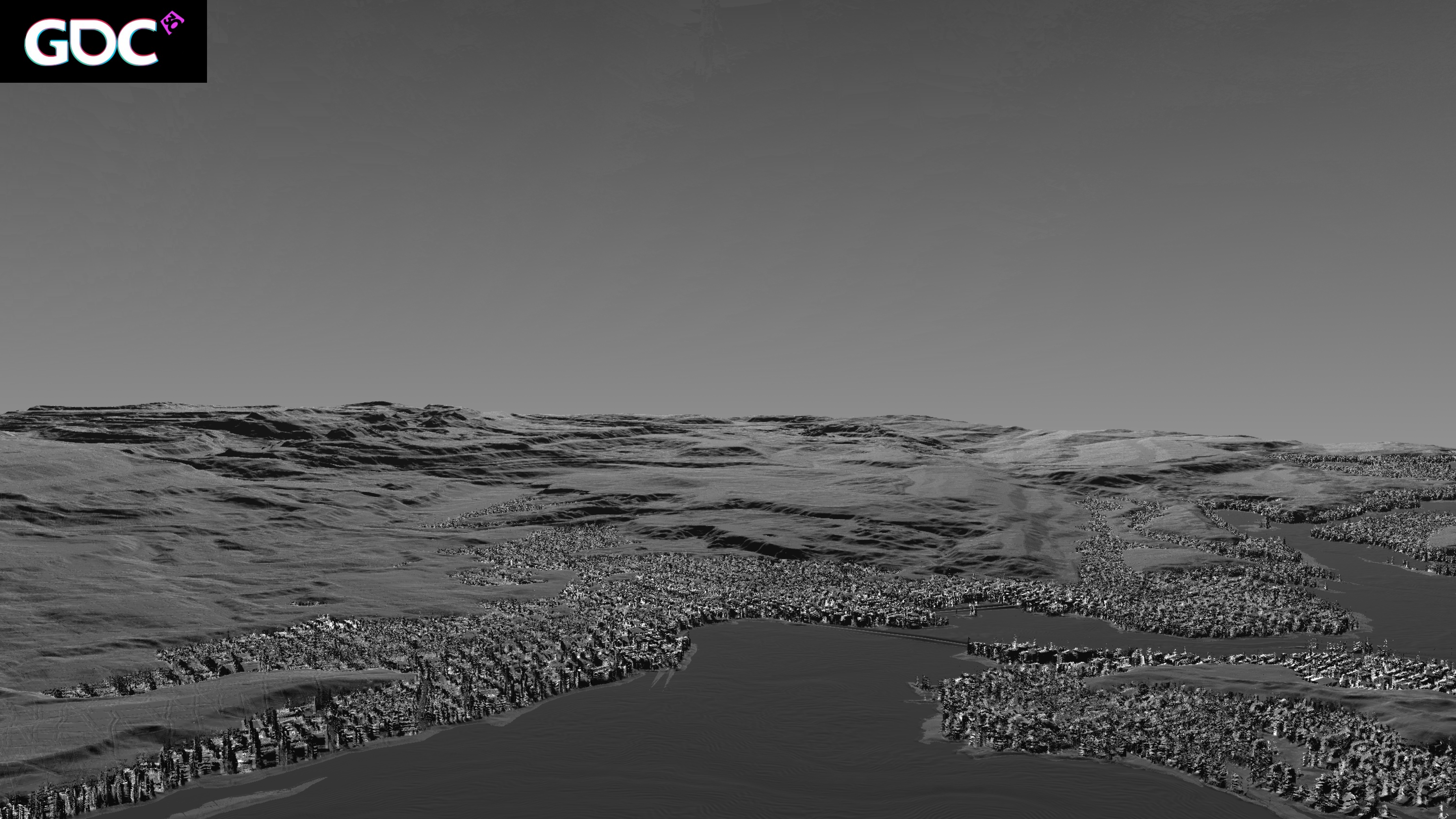





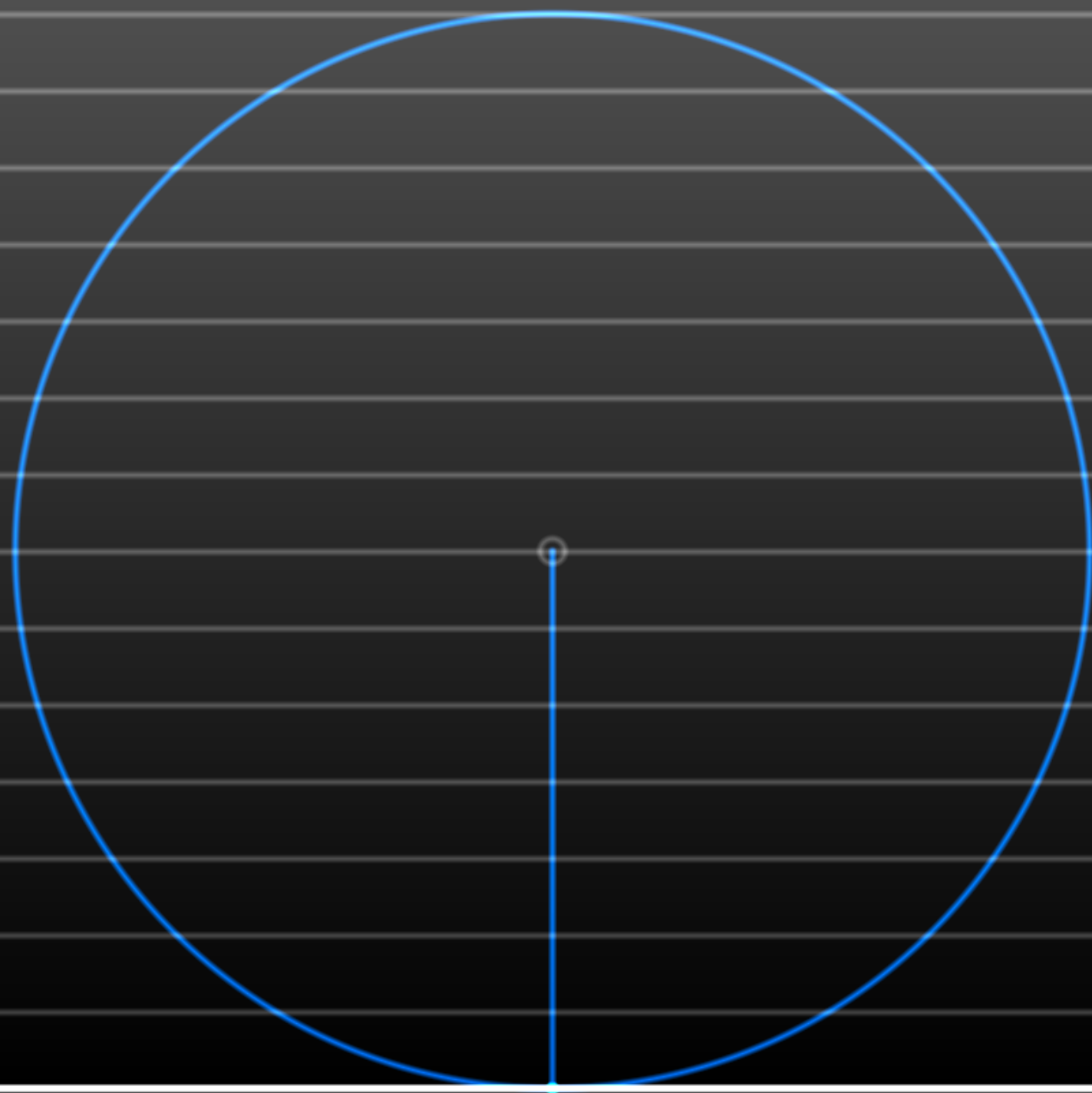
Combine these forms to create models.



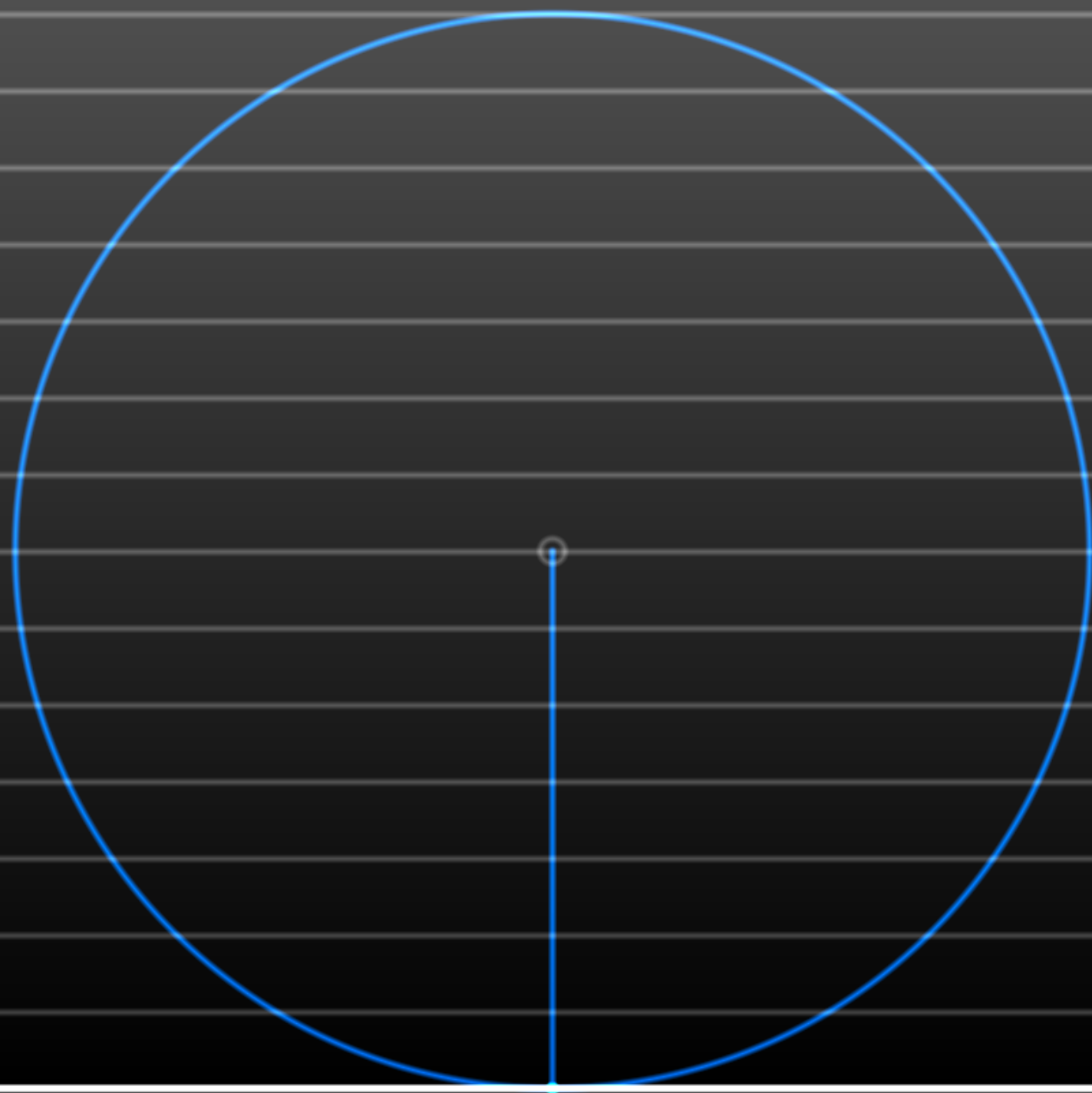




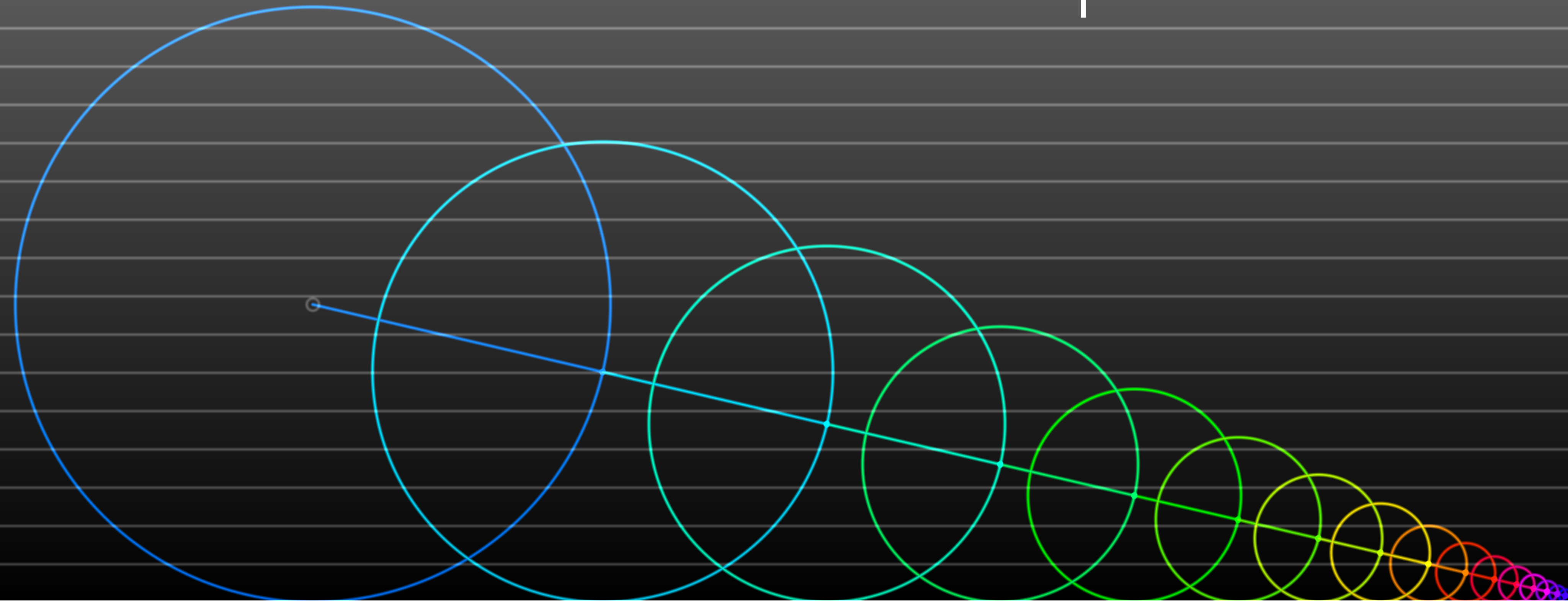
There are some things to note about distance and noise.



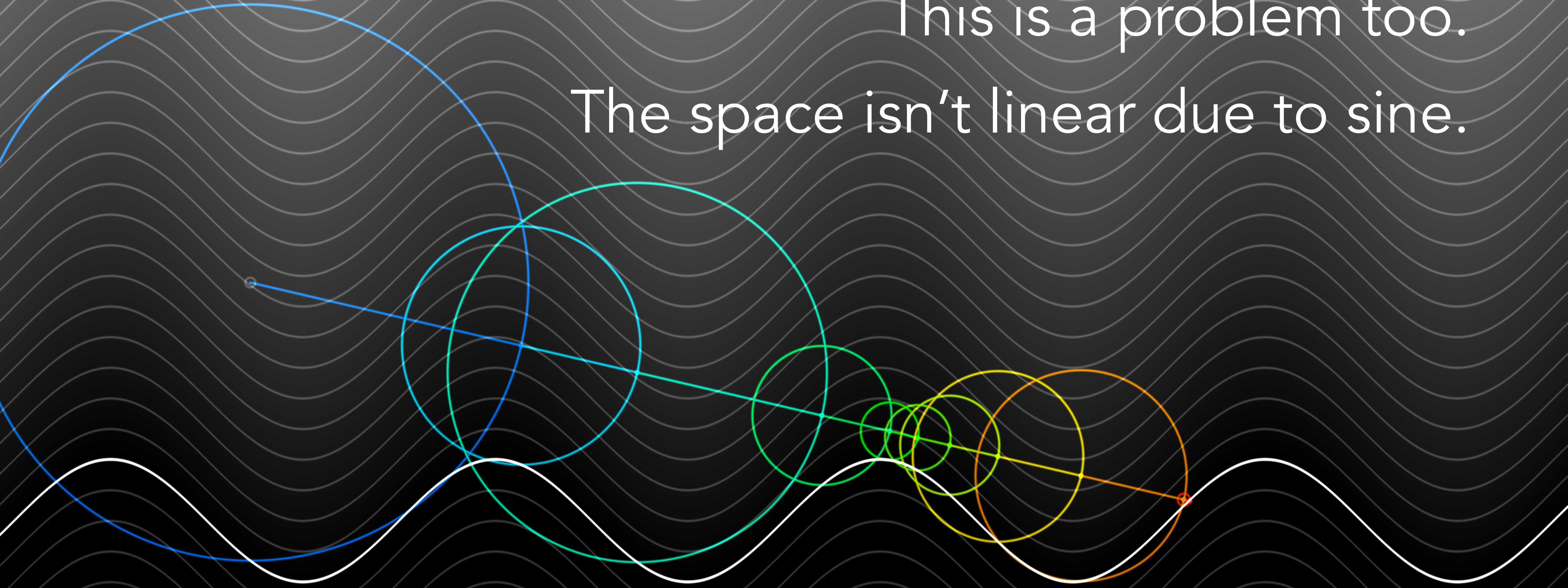
This is ideal. One test, and we find the surface.



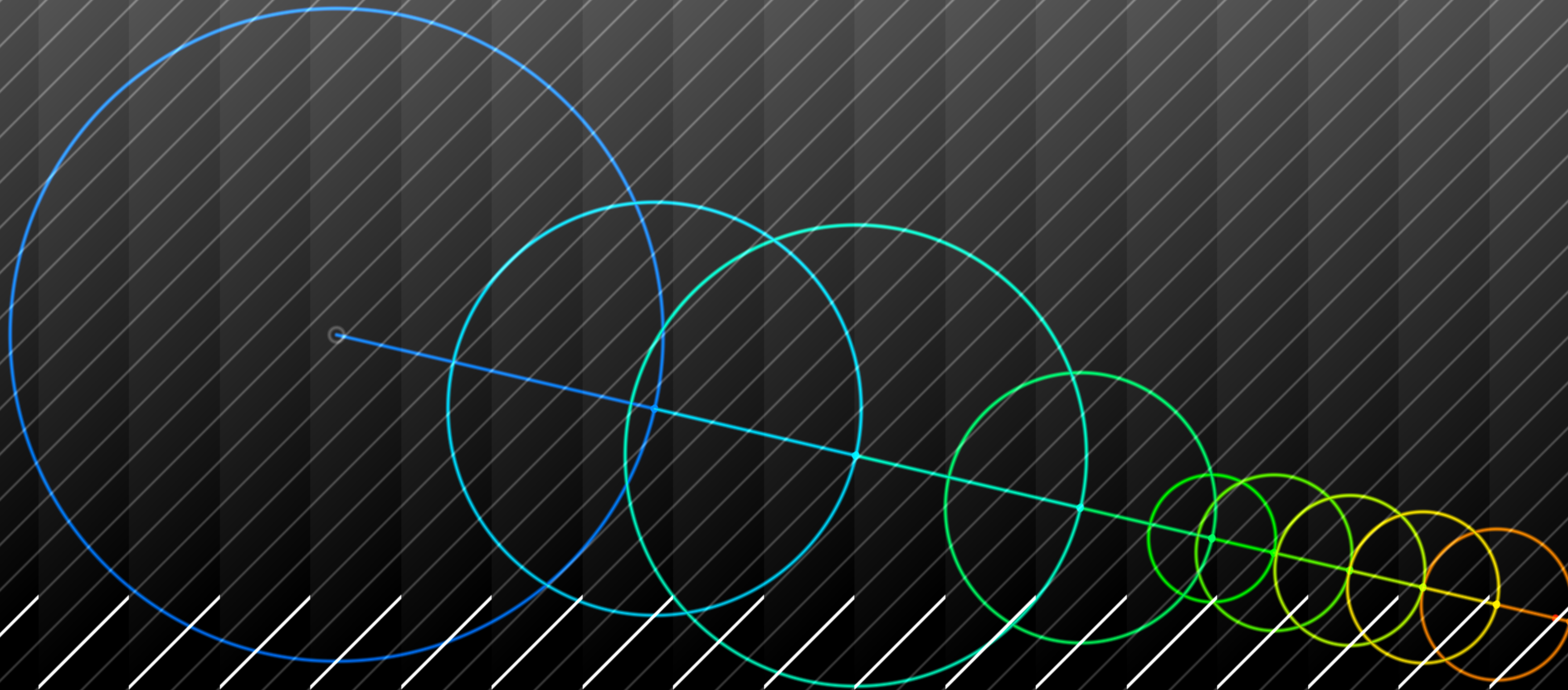
This isn't. It takes a lot of steps.



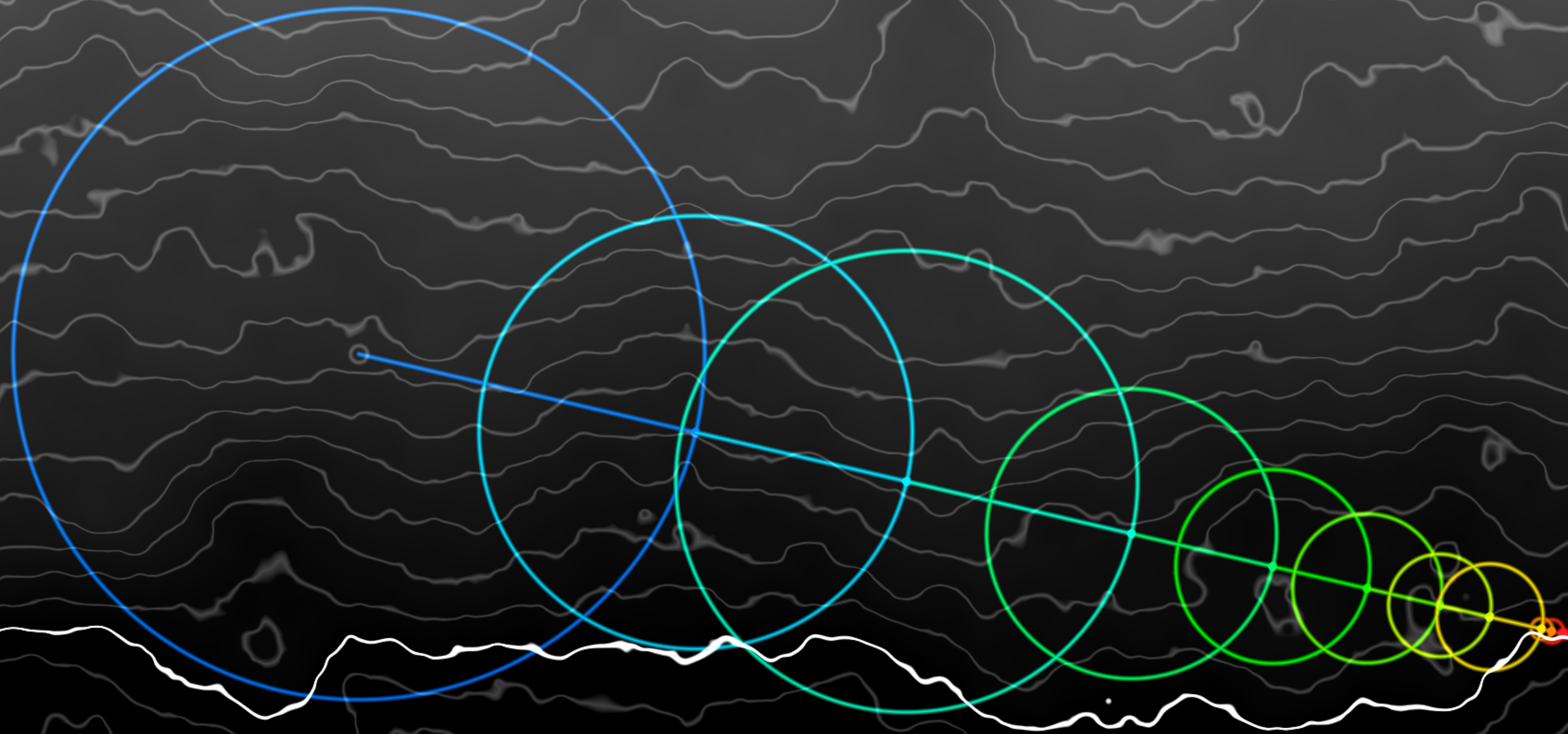
This is a problem too.
The space isn't linear due to sine.

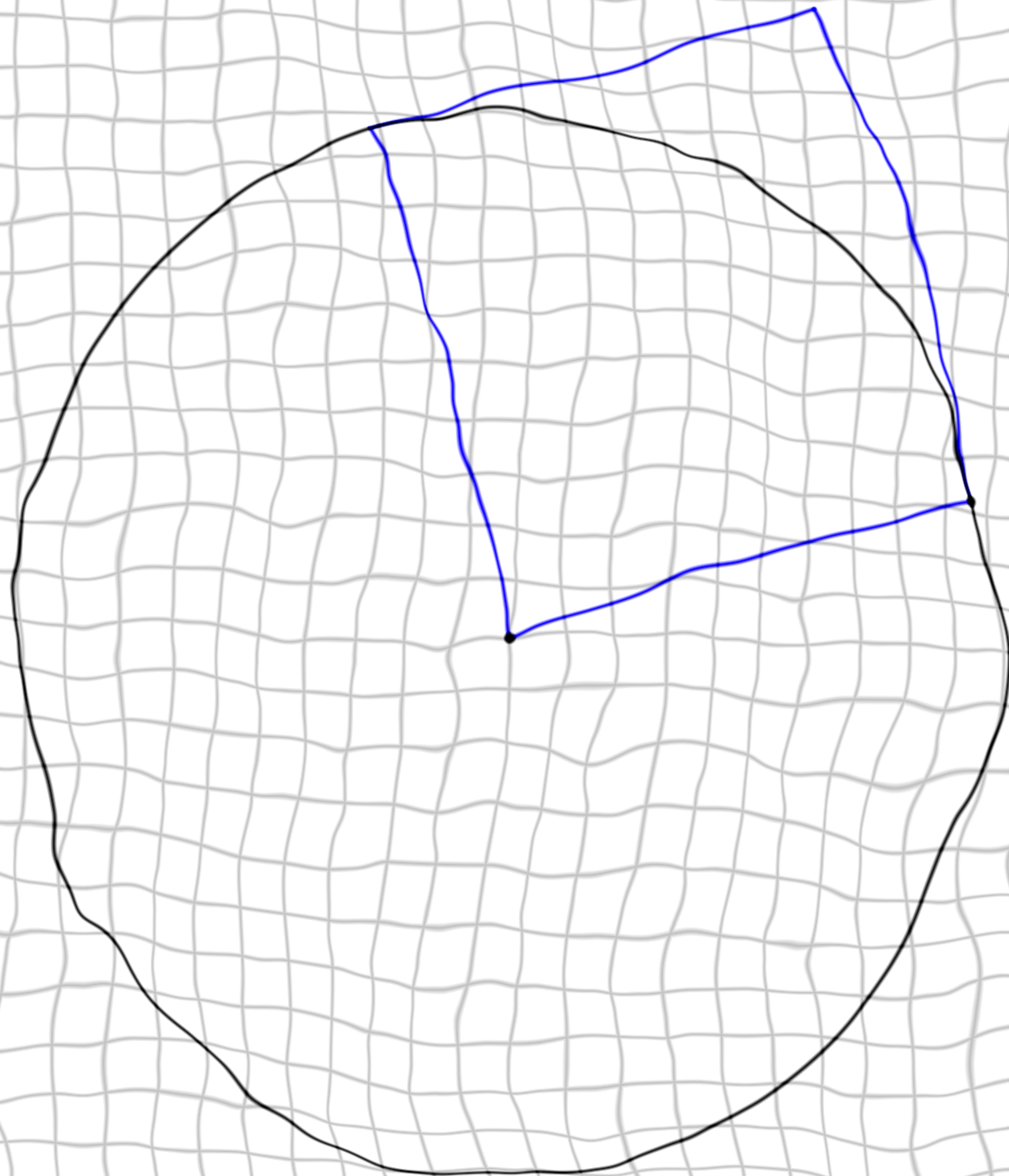


Mod creates issues as well.

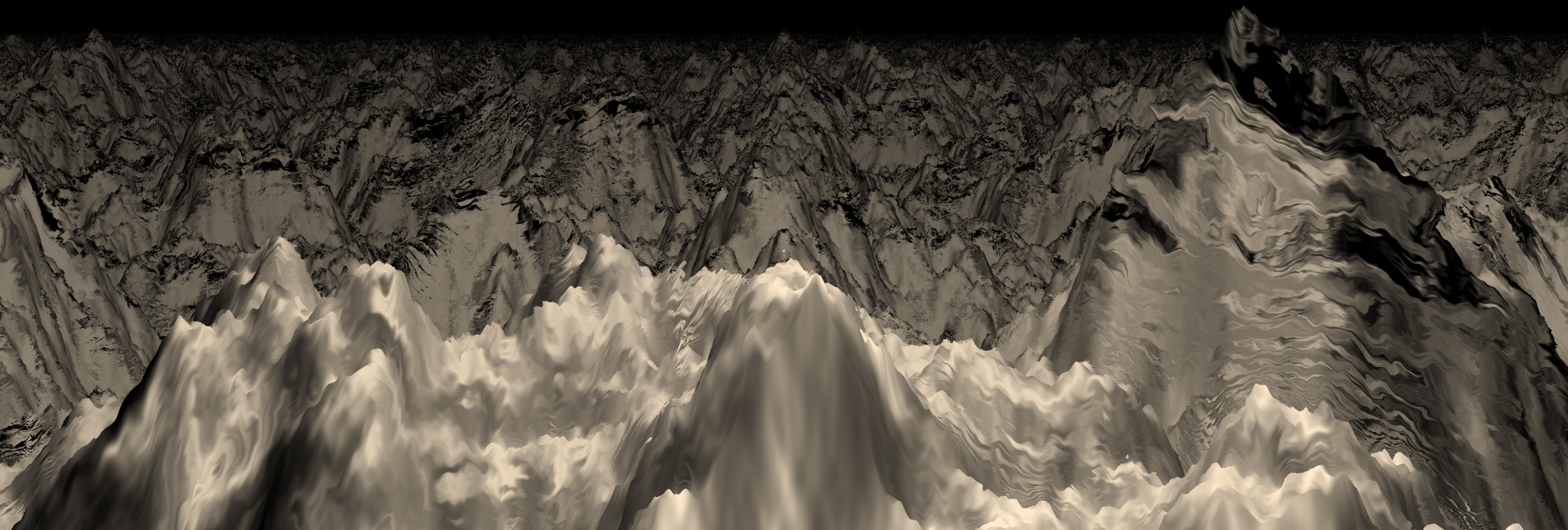


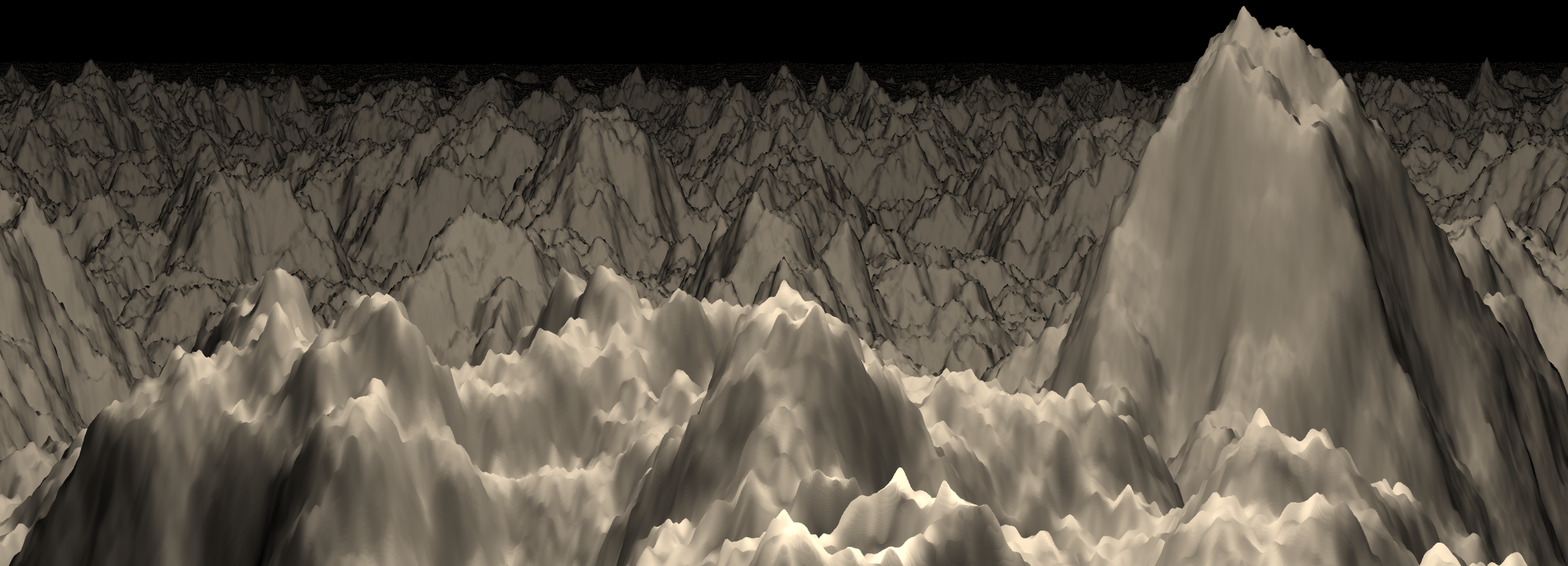
And of course, noise is noise.

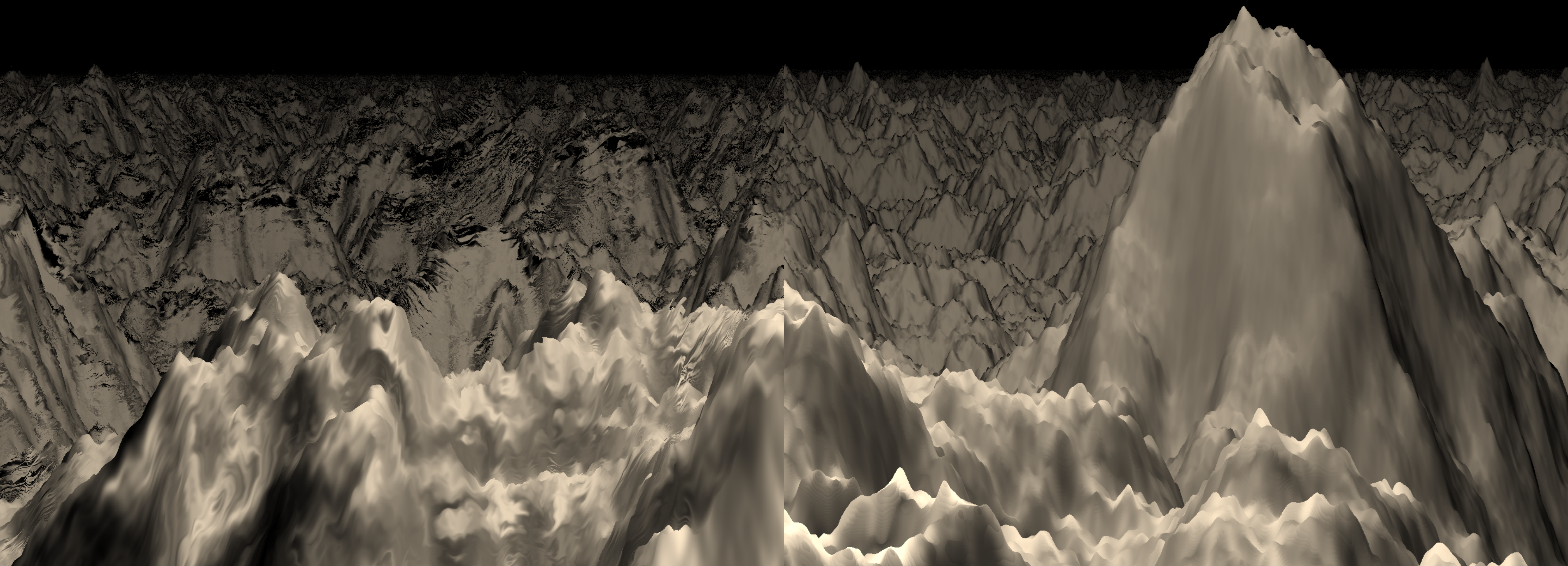


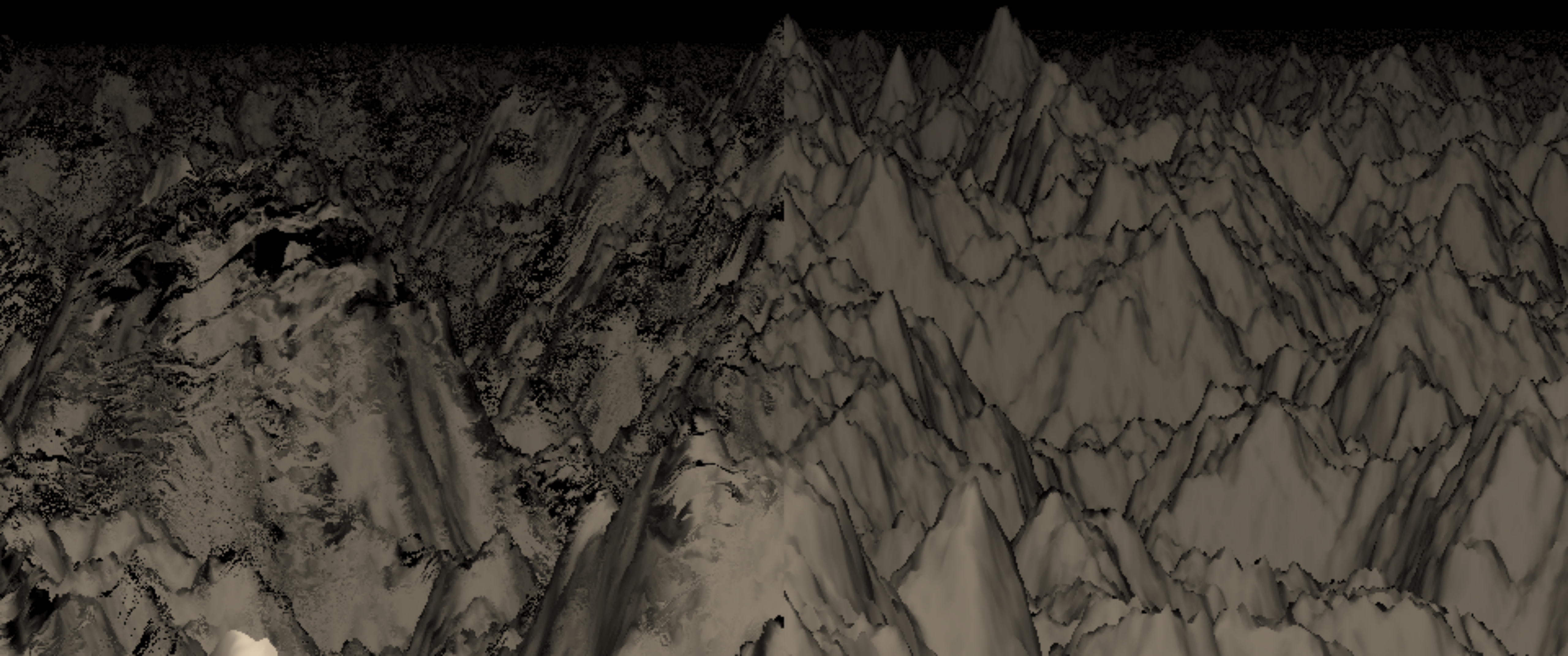


Noise rendering artifacts.









Some say you should avoid noise at all costs, as it breaks the rendering.



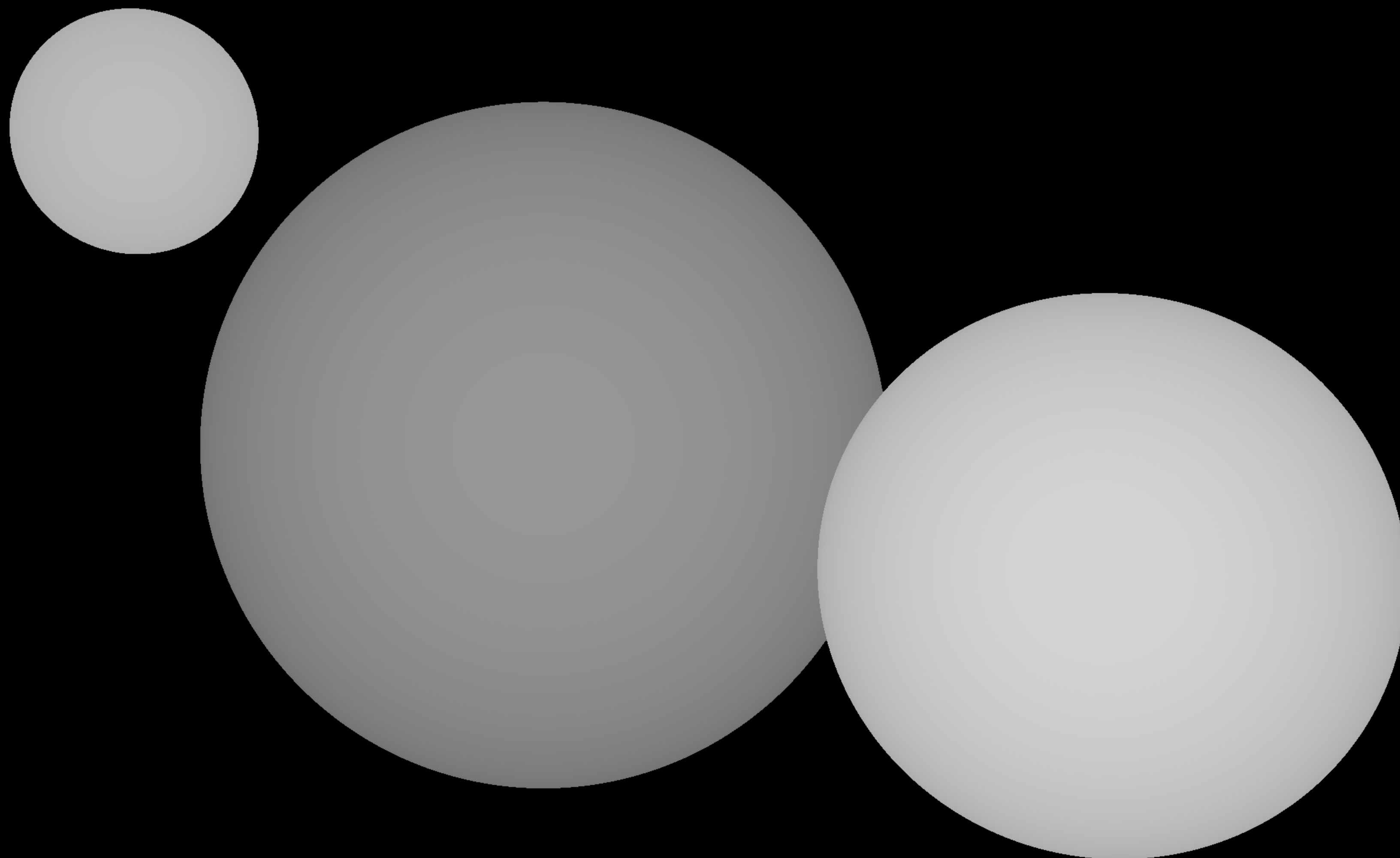




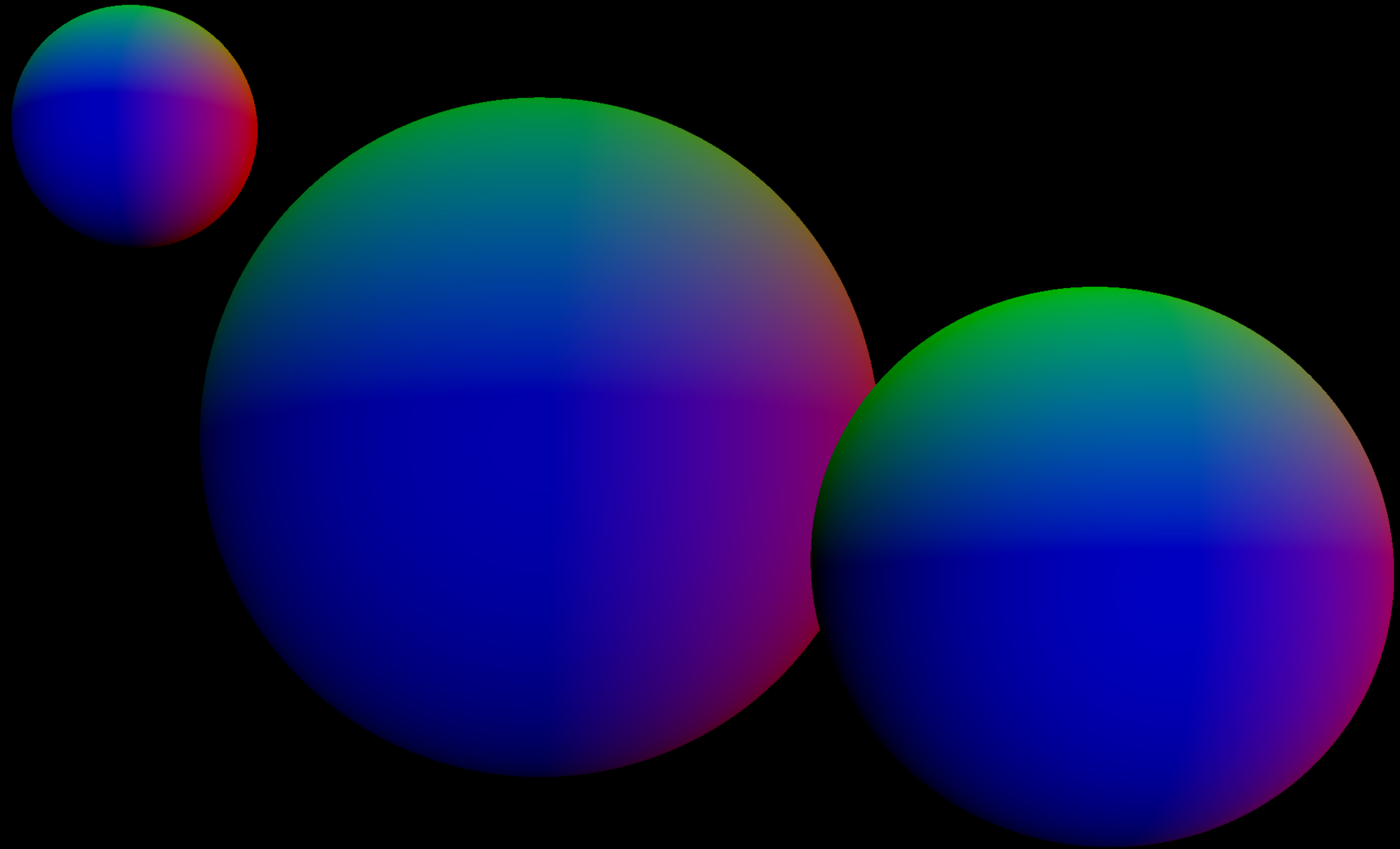
Others disagree.

Review

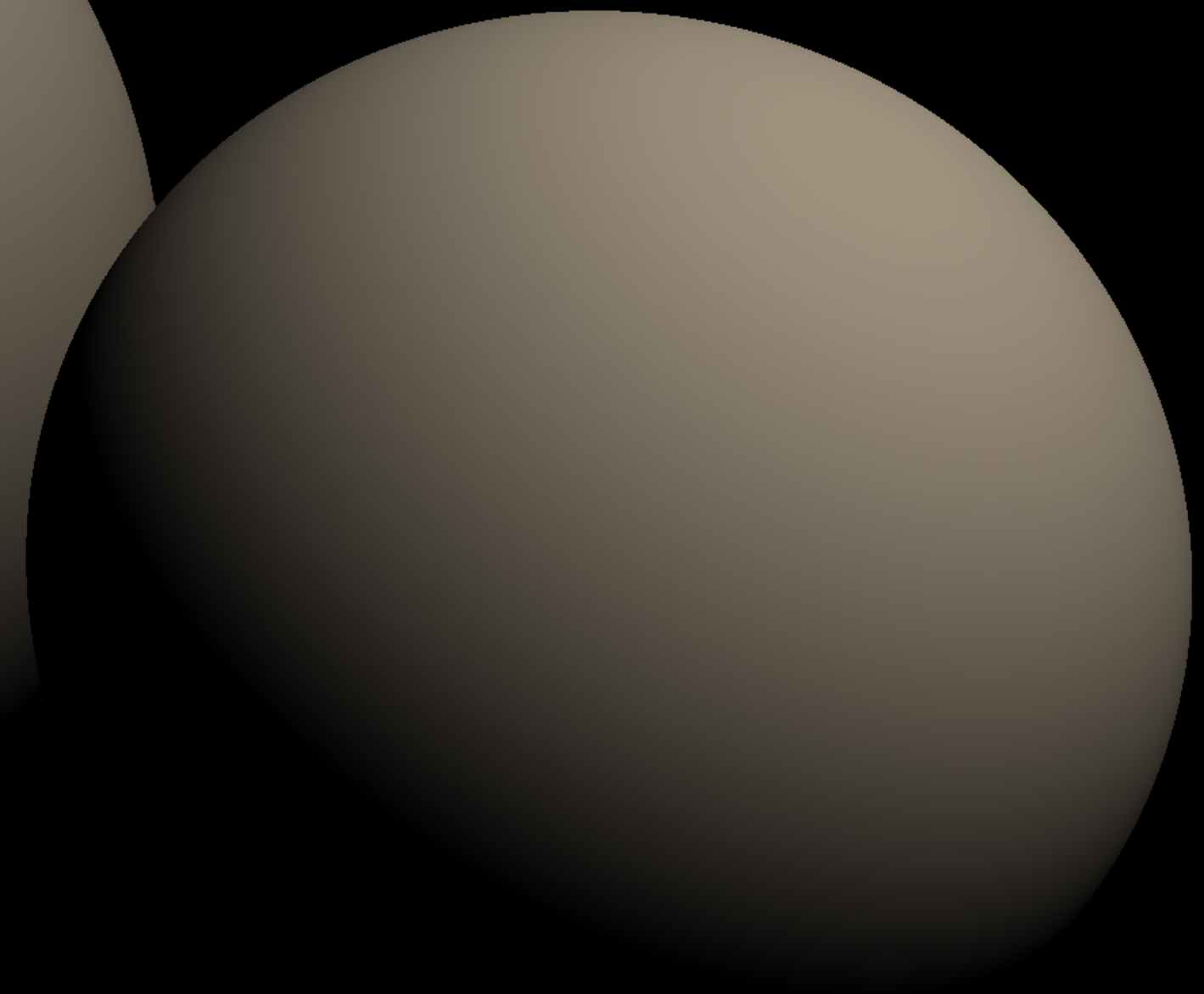
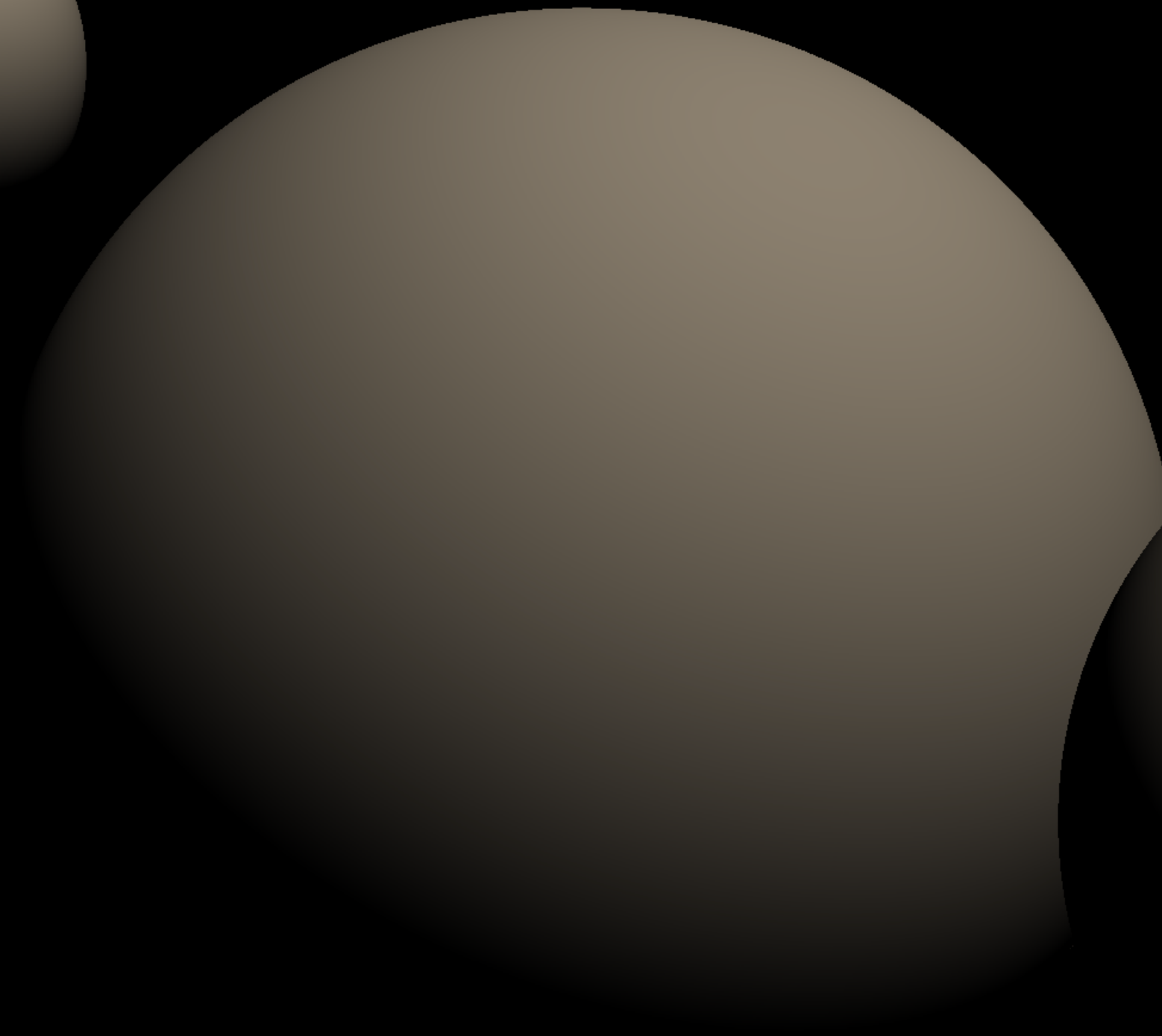
Coordinates



Depth



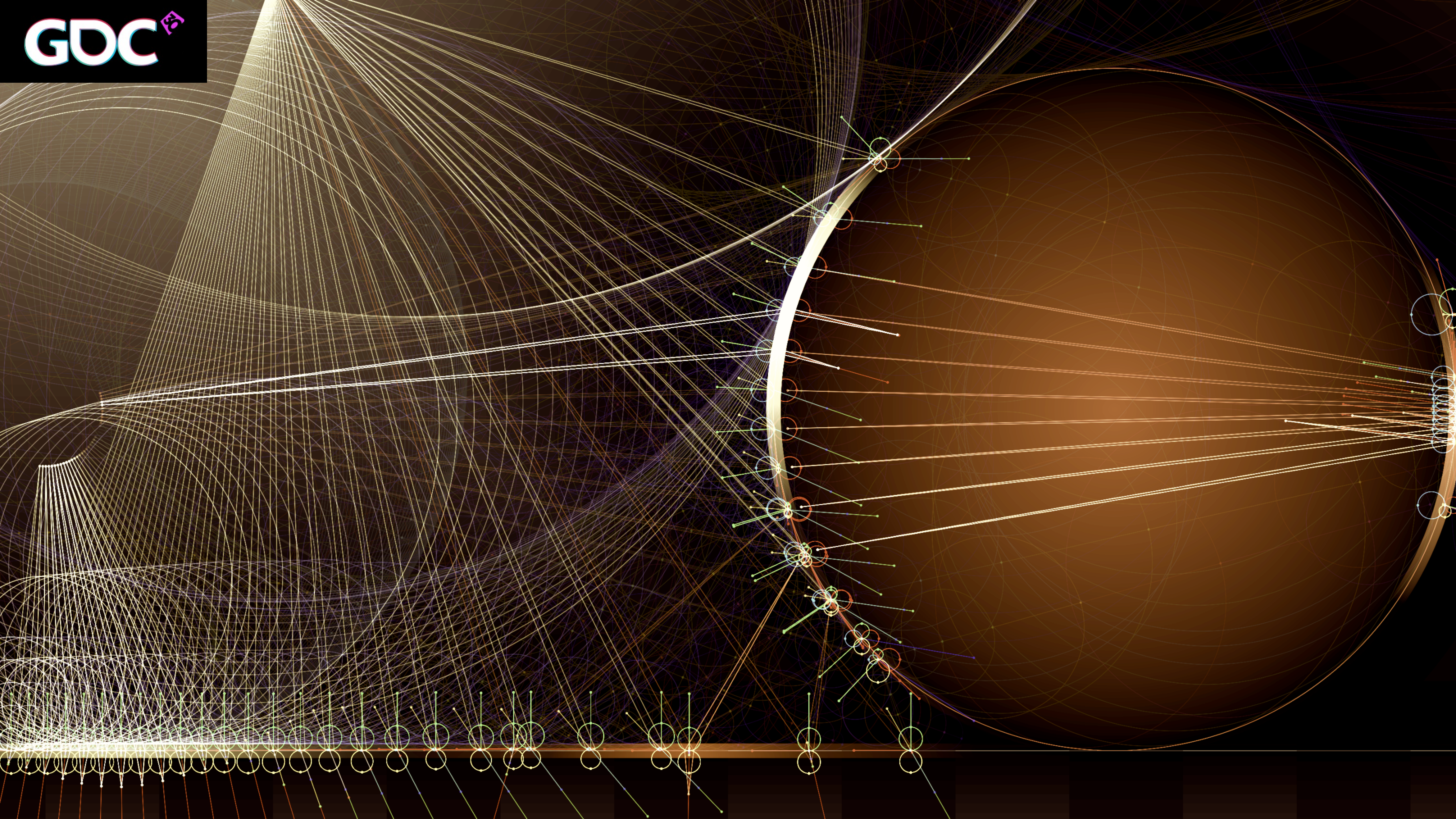
Normals

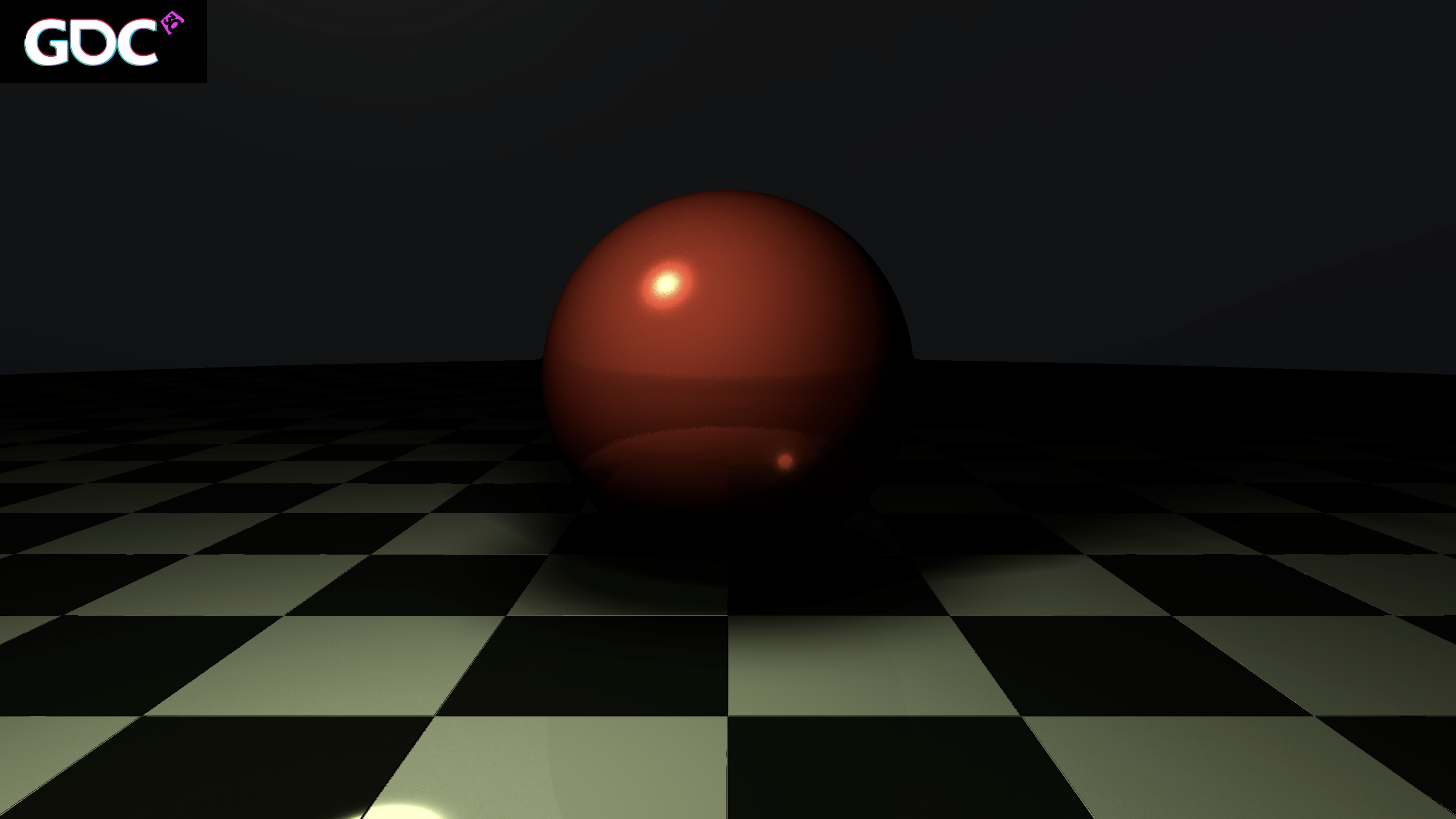


Light



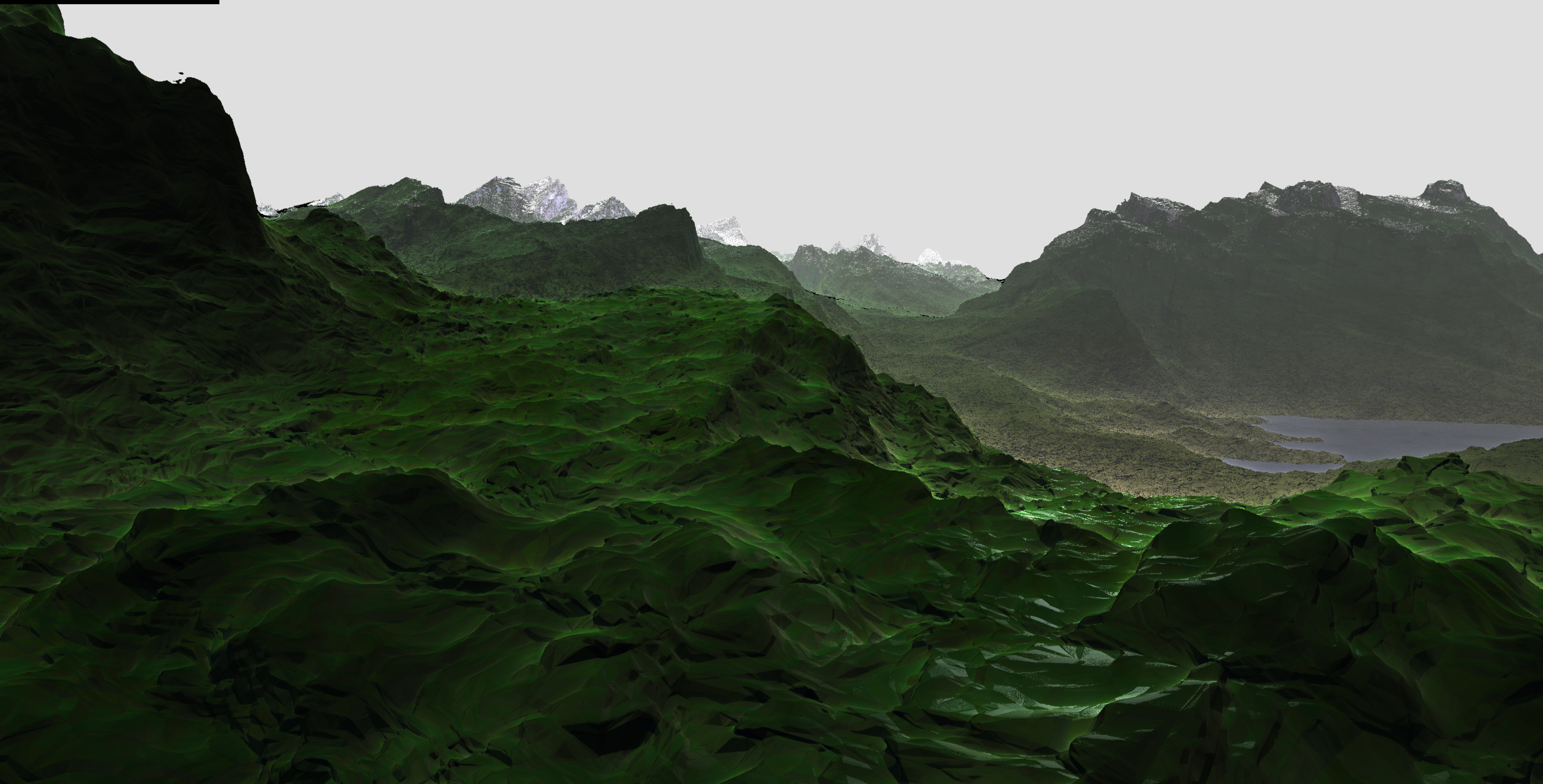
Go further.







Make your world.





Play!

Questions?

cbirke@gmail.com

[Portfolio](#)



- Live Coding Sites •

GLSL Sandbox

ShaderToy



• Demos •

[2D Spheretracer Visualization](#)

[2D Distance Fields](#)

[Basic 3D Spheretracer](#)

[Noise Example](#)

[Procedural City Example](#)

[Compass and Straightedge Model](#)

[Simple Watershed Simulation](#)

[Slides](#)



- Distance Field Modeling Resources •

hg_sdf • A library for building signed distance functions.

How to model with Distance Functions • Johann Korndorfer • NVScene 2015

Enhanced Sphere Tracing Whitepaper

Procedural Generation On Reddit



- Íñigo Quílez Resources •

[Main Site](#)

[Article on Distance Functions](#)

[Demo Of 3D Distance Functions on ShaderToy](#)

- formulanimations tutorials •

[Writing a Basic Raytracer](#)

[Making a Snail](#)



• Demoscene Links •

[Quiet Steps by keyboarders • Kindergarden](#)

[Conspiracy • Binary Flow](#)

[Digital Dynamite • Tilesapes | Demoscene](#)

[The butterfly effect • Andromeda Software Development](#)

[Photon • Firelight](#)

[Intrinsic Gravity • Still](#)

[Pouet.net](#)

